CDER eBOOK • First Edition • 2026.

Toward Modern Models Of Introductory Computing Courses

CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts



Prasad • Weems • Thota • Sussman • Vaidyanathan
Gannod • Crockett • Bunde • Spacco
Srivastava • Bourke • Suo • Maher • Gruner • Smith • Zhu • Wang

nsfexemplar.cdercenter.org

Toward Modern Models of Introductory Computing Courses

CS1 and CS2 Course Exemplars Infused with Parallel and Distributed Computing Concepts[1]

Editorial and Backbone Team

Prasad, Sushil K. (Project Coordinator; University of Texas at San Antonio),
Weems, Charles and Thota, Neena (University of Massachusetts Amherst),
Sussman, Alan (University of Maryland, College Park),
Vaidyanathan, Ramachandran (Louisiana State University)

Development Teams

Gannod, Gerald and Crockett, April (Tennessee Technological University),
Bunde, David and Spacco, Jaime (Knox College)

Testing Teams

Srivastava, Srishti (University of Southern Indiana),
Bourke, Chris (University of Nebraska–Lincoln),
Suo, Xiaoyuan and Maher, Peter (Webster University),
Gruner, Charlotte (Casper College),
Smith, Mary (Hawai'i Pacific University),
Zhu, Michelle (Kennesaw State University) and Wang, Jiayin (Montclair State University)

https://NSFexemplar.CDERcenter.org/

[1]NSF Award #2321015; **Contact:** contact@nsfexemplar.cdercenter.org, sushil.prasad@gmail.com.

Join the discord: https://discord.gg/xdh3uvD3b

How to cite this e-book: Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the ebook authors. Unless otherwise noted, this work is made available for educational use with attribution.

Contents

0 Overview and Roadmap: *Project Rationale, Methods, and Adoption Pathways* **4**
Sushil K. Prasad, Alan Sussman, Neena Thota, R. Vaidyanathan, and Charles Weems

PART I - Computer Science 1 (CS1) **27**

1 Common CS1 Parallel and Distributed Computing Activities and Lessons: *Shared Activity Families, Adoption Guidance, and Evidence* **28**
April Crockett, Srishti Srivastava, and the NSF Exemplar Project Team

2 CS1 Course Package – Tennessee Technological University: *Large-Section C++ Infusion with Unplugged Activities, OpenMP, and Remote Data* **65**
April R. Crockett and Gerald C. Gannod

3 CS1 Course Package – Knox College: *Java-Based Liberal-Arts Adaptation with Flag Maker, Greenfoot, and Knoxcraft* **105**
David P. Bunde and Jaime Spacco

4 CS1 Course Package – University of Southern Indiana: *Structured Java Activity Adoption with Evidence-Rich Unplugged and Plugged-In Modules* **123**
Srishti Srivastava

5 CS1 Course Package – University of Nebraska–Lincoln: *Scalable Codeless and Low-Code PDC Modules for Large and Online CS1 Contexts* **163**
Chris Bourke

6 CS1 Course Package – Webster University: *Visualization-First C++ Infusion through Animations, Simulations, and Game-Based Activities* **179**
Xiaoyuan Suo and Peter Maher

7 CS1 Course Package – Casper College: *Community College Adoption across Unplugged, OpenMP, Remote-Data, and Physical-Computing Activities* **197**
Charlotte Gruner

8 CS1 Course Package – Hawai'i Pacific University: *Low-Preparation Unplugged and Plugged-in PDC Activities for Small Classes* **229**
Mary L. Smith

9 CS1 Course Package – Montclair State University: *Focused Java Minimal-Infusion Model Centered on Flag Maker and Data Parallelism* **267**
Jiayin Wang and Michelle Zhu

PART II - Computer Science 2 (CS2) **287**

10 Common CS2 Parallel and Distributed Computing Activities and Lessons: *Shared Activity Families, Adoption Guidance, and Evidence* **288**
David Bunde, April Crockett, and the NSF Exemplar Project Team

11 CS2 Course Package – Tennessee Technological University: *Large-Section C++ Infusion with Plugged-In Activities using OpenMP* **320**
April R. Crockett and Gerald C. Gannod

12 CS2 Course Package – Knox College: *Java-Based Liberal-Arts Adaptation using the Fork-Join Framework* **343**
David P. Bunde and Jaime Spacco

13 CS2 Course Package - University of Southern Indiana: *Structured Java Infusion through Event-Driven Programming, Fork–Join, and Parallel Image Processing* **357**
Srishti Srivastava

14 CS2 Course Package - University of Nebraska–Lincoln: *Minimal-Change Java Integration through Connection Pooling and Parallel Sorting* **404**
Chris Bourke

15 CS2 Course Package - Webster University: *C++ Infusion through Parallel Merge Sort, OpenMP, and Performance Analysis* **414**
Peter Maher

16 CS2 course package using Python - Webster University: *Python-Based Infusion through Multiprocessing, Parallel Image Processing, and Performance Engineering* **440**
Xiaoyuan Suo

17 CS2 Course Package – Casper College: *Community College Adoption across Unplugged, OpenMP, and Greenfoot Activities* **457**
Charlotte Gruner

18 CS2 Course Package – Montclair State University: *Java Infusion through Event-Driven Greenfoot Game and Parallel Image Processing* **481**
Michelle Zhu and Jiayin Wang

Bibliography **509**

Chapter 0

Overview and Roadmap

Project Rationale, Methods, and Adoption Pathways[†]

Sushil K. Prasad[1], Alan Sussman[2], Neena Thota[3], Ramachandran Vaidyanathan[4] and Charles Weems[3]

[1]University of Texas, San Antonio
[2]University of Maryland, College Park
[3]University of Massachusetts, Amherst
[4]Louisiana State University, Baton Rouge

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 6
0.1 Introduction . . . 8
0.2 Project Focus and Team Structures . . . 10
0.3 Methodology Employed . . . 13
0.3.1 PDC Topics Chosen: Data Parallelism, Distributed Data Access, and Event Driven Processing . . . 13
0.3.2 Testing Team Recruitment . . . 14
0.3.3 Collaboration among Development and Testing teams, and support of Backbone team . . . 14
0.3.4 Exemplar Creation and Testing by the Development Teams . . . 15
0.3.4.1 TTU CS1 and CS2 Exemplar Development (C++) . . . 15
0.3.4.2 Knox CS1 and CS2 Exemplar Development (Java) . . . 15
0.3.5 Adaptation and Adoption by Testing Teams . . . 16
0.3.6 Evaluation Methods . . . 16
0.4 Faculty Learning Community Formation . . . 17
0.5 Roadmap for Subsequent Chapters . . . 18
0.5.1 Start from Your Adoption Goal . . . 18
0.5.2 CS1 and CS2 Exemplars at a Glance . . . 18
0.5.3 Using the Common Chapters with Institutional Exemplars . . . 21
0.5.4 Finding Materials Quickly . . . 22
0.5.5 Summary Guidance and Conclusions . . . 23
Appendix . . . 24
Acknowledgements . . . 26

Chapter figures

1 Project activities and Timeline. . . . 9
2 Suggested reading paths by course and scale of adoption. . . . 19
3 Organization of Chapters 1 and 10 – common CS1 and CS2 activities and adoption guidance. . . . 21
4 Common structure of institutional exemplar chapters. . . . 22

Chapter tables

1 List of institutions participating in the work in this volume . . . 12
2 Quick comparison of CS1 and CS2 institutional exemplars for potential adopters. . . . 19

Abstract

Modern software is increasingly parallel, distributed, networked, event-driven, graphical, API-based, and data-intensive. However, introductory computing courses still commonly begin with a sequential-first model of computation which does not fully reflect the systems students encounter today. Students regularly use applications that depend on multicore processors, remote services, asynchronous events, large data streams, and libraries that hide complex computation. Yet many students do not learn these ideas formally until advanced electives, if at all. This gap limits their preparation for later study and work in software engineering, artificial intelligence, cybersecurity, high-performance computing, computational science and engineering, and other areas of computing and applications.

This volume responds with a practical adoption guide for the introductory computing course sequence for instructors, course coordinators, departments, and academic leaders. Rather than only arguing that introductory courses should be modernized, it provides classroom-tested, modular course materials that can be adopted at different scales. The goal is not to turn introductory computing students into parallel programmers, but to help them develop an early conceptual model in which computation may involve multiple workers, remote data, performance tradeoffs, and coordination among interacting components, all the while ensuring that the basic CS1 and CS2 learning outcomes are preserved.

This work results from an NSF-funded multi-institutional project to create modern CS1 and CS2 course exemplars infused with parallel and distributed computing (PDC) concepts that can serve as easily adoptable models.[1] To enable this vision, the project began with two initial course exemplars developed in different settings: a Java-based sequence at a small liberal arts college and a C/C^{++}-based sequence in a medium-sized computer science department in an engineering-college setting. These exemplars were then adopted, adapted, and evaluated by six carefully-recruited institutions ranging from a community college to private undergraduate institutions and an R1 university. This diversity is central to the project: no single model fits all languages, calendars, class sizes, student populations, or instructor backgrounds. This project thus emphasizes multiple pathways for PDC infusion.

This volume includes both CS1 and CS2 course exemplars that have been classroom tested and evaluated, together with shared activities and substantial sets of instructional resources, which will be periodically curated. Across these chapters, readers will find a range of approaches: short conceptual activities requiring little or no programming background; unplugged exercises that model processors, data partitioning, speedup, load imbalance, and communication; visualizations and simulations that make abstract PDC concepts visible; lightweight programming assignments that connect PDC ideas to loops, arrays, sorting, searching, APIs, and event handling; more substantial CS2 activities involving task parallelism, shared-memory programming, synchronization, fork-join, image processing, and performance measurement; and assessment instruments that capture student background, self-reported experience, engagement, and learning evidence. Together, the chapters illustrate that PDC can be introduced through existing CS1 and CS2 topics rather than treated only as additional content.

Across the sequence, CS1 generally establishes early conceptual awareness that computation need not be strictly sequential, while CS2 provides opportunities to deepen that

[1] NSF Exemplar Project Award #2321015-20

foundation through implementation, correctness reasoning, synchronization, and empirical performance analysis. At the same time, the CS2 exemplars also show how short conceptual on-ramps can be used when students enter CS2 without prior PDC exposure.

This chapter serves as a roadmap for the volume. Readers may navigate the material by programming language, activity type, class size, instructional time, institutional context, or desired level of adoption. An instructor seeking a single activity can begin with the distilled set of common activities described in Chapter 1 for CS1 or Chapter 10 for CS2, and then consult institutional examples for implementation details in the corresponding CS1 or CS2 institutional chapters, each with their supporting instructional material ready to download and adopt. An instructor seeking a module or course-level infusion can compare the institutional chapters, course schedules, assessments, and lessons learned. Some consultation from project personnel will be available to help adopters select and adapt materials. Additional instructor training workshops will be organized through a new NSF award to further infuse AI and Big Data, together with PDC, into CS1, CS2, and Computer Systems courses by the CDER center.[2]

This volume also provides additional insight to those looking for pedagogical approaches and the basis and impact of the project that led to this work.

How to Read this Volume: Depending on your primary objective, we provide some initial short cuts to relevant parts of the volume. Nevertheless, the volume in its entirety provides a perspective and depth that may not be available from a selective reading.

- Instructor seeking to include new material: Start from Section 0.5 and then move to Chapter 1 for CS1, Chapter 10 for CS2, and relevant chapters as indicated in those sections.
- Researcher looking for pedagogical approaches and research results start from Section 0.3.
- Those looking to understand the basis and impact of this project continue to read this chapter.

[2] https://CDERcenter.org/

0.1 Introduction

The CDER center organized a series of stakeholder workshops during 2020–23, supported by a National Science Foundation (NSF) CyberTraining Institute Conceptualization grant[3] that examined the systemic barriers to broader PDC adoption, including the continued dominance of a sequential model of computation in early courses. Multiple employer stakeholders including national labs and agencies reported that computer science and engineering students are not graduating with a sufficient grasp of the software development process for the modern computing ecosystem, which relies on parallelism, distribution, asynchrony, scaling, integration across disparate libraries and data sources, test-driven design, and pervasive concerns for security. The lack of preparation in parallel and distributed computing (PDC) is rooted in an obsolete algorithmic model of computing (sequential, synchronous, with text-based I/O) that instruction relies upon, from introductory programming to data structures, algorithms, and software engineering. Once this outdated model is entrenched, students may only encounter PDC in an *ad hoc* manner via electives. Education stakeholders said that changing the model is nearly impossible because of a lack of exemplars of how introductory courses can be modernized, few teaching materials, a lack of instructor training in modern computing, and little evidence of how changing the model benefits students. Perhaps the greatest impediment is that academic stakeholders find it difficult to visualize what a different approach would look like, and most feel it is not possible to make such a change. What we heard repeatedly is that "someone else needs to go first."

Project Overview

This volume is a result of our followup work on a NSF-funded multi-institutional collaborative project "Modern Course Exemplars infused with Parallel and Distributed Computing for the Introductory Computing Course Sequence (2023–2027)." This project adopts a "go first" effort to create exemplars of first year course sequence (typically CS1 and CS2) changes and accompanying materials, while demonstrating their effectiveness and transferability. The project employed two teams to develop these exemplars in two distinct contexts, and then trained a cohort of instructors at six additional institutions to implement them, along with instrumenting their interventions so that student outcomes can be documented. A backbone team helped coordinate the various moving parts of the project. Figure 1 captures the overall project activities and timeline. The result is a demonstration of the potential for modernizing the computing model that shapes how first-year computing students think computationally, enabling them to conceptualize how software can make use of parallelism and distribution.

An important feature of the project is that CS1 and CS2 were treated as a connected introductory sequence. CS1 ideas were developed first, and experience with CS1 informed the development and adaptation of CS2. In general, CS1 provided early conceptual exposure to parallelism, distribution, events, coordination, and performance tradeoffs, while CS2 created opportunities for deeper implementation, correctness reasoning, synchronization, and empirical performance analysis. Where students entered CS2 without prior PDC exposure,

[3] NSF Institute Conceptualization Award #2002649,#1923810

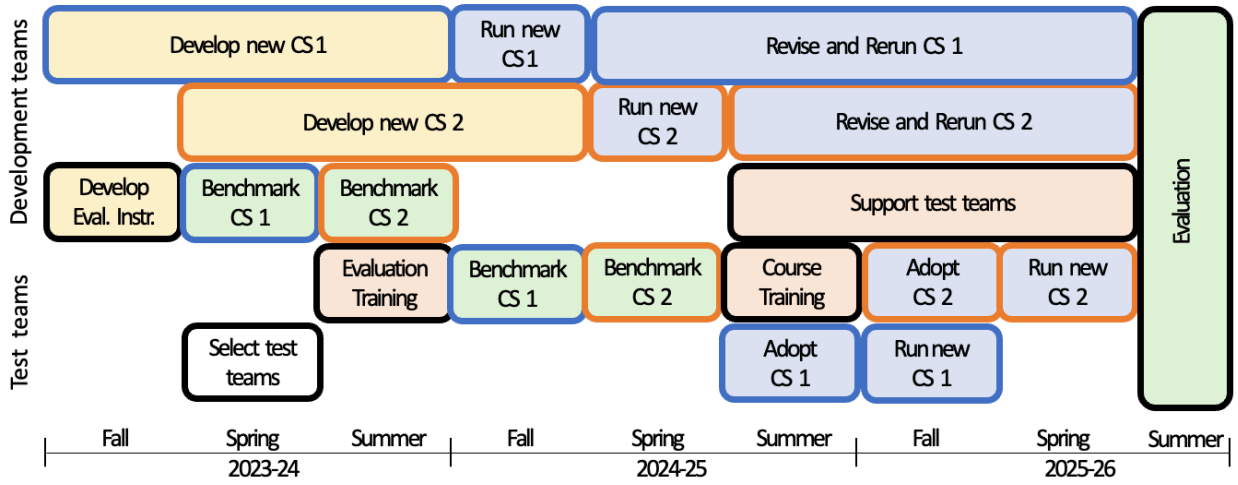


Figure 1: Project activities and Timeline. Box fill colors categorize activities as development (yellow), evaluation (light green), training and support (peach), and running the new courses (blue).

introductory conceptual activities could also serve as an on-ramp.

Along the way, many opportunities and challenges were identified. Are concepts in PDC too advanced at the early course level? On this we found *unplugged activities* which entail no programming, but underscore key concepts, worked well in multiple implementations. Is there room for introducing new material in an already crowded course? We found it was possible to teach traditional concepts with a different focus; for example, a lesson on loops could be augmented (with a small effort) to include a *Flag Maker activity* wherein students act as processors coloring flag-grid cells in sequential and parallel scenarios. Similar opportunities appear in CS2: sorting, recursion, data structures, event handling, image processing, and performance analysis can provide natural anchors for introducing task decomposition, shared-memory parallelism, synchronization, and speedup without turning CS2 into a separate parallel-programming course.

Key Contributions of this Volume: This volume captures the significant findings, observations and deliverables from the project described above. Specifically,

- It describes, through institutional CS1 and CS2 course-package chapters, multiple ways of infusing PDC across the introductory computing sequence. The authors of these chapters describe their classroom-based evaluation, adoption and adaptation experiences, and an appendix for supporting instructional material in a GitHub.
- It distills a set of popular unplugged and programming activities alongside infusion methods and challenges faced into Chapter 1 for CS1 and Chapter 10 for CS2.
- It illustrates how PDC learning can be threaded across the sequence, from conceptual awareness and lightweight experiences in CS1 toward implementation, synchronization, correctness reasoning, and performance analysis in CS2, while also supporting CS2 adoption when students have had little or no prior PDC exposure.
- It provides a bird's-eye view of the project activities that led to this e-book, including team structures, methodology employed, learning community formation, a road map for readers, and the backdrop of prior work (this chapter).

Chapter organization: The remainder of this chapter is organized as follows. Section 0.2 describes the project focus, the development and testing teams, and the role of the backbone team. Section 0.3 explains the PDC topics selected for infusion, the recruitment of testing teams, the collaboration process, and the evaluation methods used across institutions. Section 0.4 discusses faculty learning community formation and sustainability. Section 0.5 provides an adopter-oriented roadmap for navigating the common CS1 and CS2 chapters, the corresponding institutional chapters, appendices, and repositories. The appendix to this chapter situates this project within the broader CDER center activities, including IEEE-CS TCPP curriculum, early-adopter, training, and dissemination efforts, that shaped the work reported in this volume.

0.2 Project Focus and Team Structures

As noted above, the project targeted the CS1, CS2 sequence of a typical Computer Science program for the introduction of PDC concepts. For both courses, data was obtained both from a baseline offering (without any interventions) and from an offering after the PDC ideas were introduced. The teams were also introduced to methods and instruments for evaluating the efficacy of the interventions. In general, CS1 ideas were developed first [12] and experience with CS1 informed our approach to CS2. The initial organizing PDC themes can be summarized as follows:

- Data parallelism and its applications
- Distributed data access
- Event-driven processing

As the exemplars developed across the CS1–CS2 sequence, these themes were expressed at different levels of depth. CS1 generally emphasized conceptual exposure and lightweight applications, while CS2 extended the treatment to ideas such as task decomposition and

fork–join parallelism, shared-memory programming, synchronization and race conditions, work and span, speedup and parallel overhead, benchmarking, and empirical performance analysis. Thus, the three themes above provided a common starting point rather than a limit on the PDC concepts represented in the completed exemplars.

The project was implemented by three sets of teams that worked closely together.

Development Teams: These teams form the frontline in the implementation and testing of new interventions in their introductory computing courses that introduce PDC concepts. The Development Teams, with faculty from two different types of institutions and using two different programming languages, each created exemplars for their CS1 and CS2 computing courses, such that the second course builds on the concepts introduced in the first. These teams coordinated their efforts so that a common set of PDC themes was addressed, although in ways that are specific to each of their contexts. After a trial run of the new material by the development teams, they worked with the Testing Teams (see below) to transfer their exemplars to six other institutions. The experiences of the Testing Teams were then used to improve the work of both sets of teams.

There were two Development Teams, one at Tennessee Technological University and one at Knox College. Details of these teams are listed here:

1. Tennessee Technological University (TNTECH) is a public research university in Cookeville, Tennessee that offers undergraduate and graduate degrees in Computer Science (the department associated with this project). The institution operates on a semester calendar.
   - Team members: April Crockett, Gerald Gannod
   - Language used: C++
2. Knox College (Knox) is a private liberal arts college in Galesburg, Illinois that offers an undergraduate degree in Computer Science (again the department associated with this project). The institution operates on a calendar with three terms (Fall, Winter, and Spring). Classes are shortened, but students only take three courses at a time and class meetings are longer so the contact hours are roughly the same as a semester-long class.
   - Team members: David P. Bunde, Jaime Spacco
   - Language used: Java

Testing Teams: These teams worked to transfer the exemplars introduced by the Development Teams into their own courses, in the process identifying opportunities and challenges. The input from the Testing Teams was a key element in improving the teaching materials developed by the Development Teams.

Six Testing Teams described below worked on the project.

1. **Casper College (Casper)** is a public community college in Casper, Wyoming.
   Team member: Charlotte Gruner *Programming language:* C++, Java, Python
2. **Hawai'i Pacific University (HPU)** is a private university in Honolulu, Hawai'i.
   Team member: Mary Smith *Programming language:* Java

3. **Montclair State University (MSU)** is a public university in Montclair, New Jersey.
   Team members: Jiayin Wang and Michelle Zhu (now at Kennesaw State University)
   Programming language: Java
4. **University of Southern Indiana (USI)** is a public university in Evansville, Indiana.
   Team member: Srishti Srivastava *Programming language:* Java
5. **University of Nebraska, Lincoln (UNL)** is a public land-grant university in Lincoln, Nebraska.
   Team member: Chris Bourke *Programming language:* C, Java
6. **Webster University (Webster)** is a private university with a main campus in Webster Groves, Missouri.
   Team members: Xiaoyuan Suo and Peter Maher *Programming language:* C++, Python

Backbone Team: This team coordinated the project, organizing training workshops and biweekly meetings, facilitating discussion, and providing domain expertise in PDC. The Backbone Team members are

- Sushil K. Prasad, University of Texas, San Antonio
- Alan Sussman, University of Maryland, College Park
- Neena Thota, University of Massachusetts, Amherst
- Ramachandran Vaidyanathan, Louisiana State University, Baton Rouge
- Charles Weems, University of Massachusetts, Amherst

For the remainder of this volume we use the abbreviations listed in Table 1 to refer to the participating institutions.

Table 1: List of institutions participating in the work in this volume

Institution	Abbreviation
Tennessee Technological University	TNTECH
Knox College	Knox
Casper College	Casper
Hawai'i Pacific University	HPU
Montclair State University	MSU
University of Southern Indiana	USI
University of Nebraska, Lincoln	UNL
Webster University	Webster
Louisiana State University, Baton Rouge	LSU
University of Massachusetts, Amherst	UMass
University of Maryland, College Park	UMD
University of Texas, San Antonio	UTSA

0.3 Methodology Employed

0.3.1 PDC Topics Chosen: Data Parallelism, Distributed Data Access, and Event Driven Processing

When instructors hear about incorporating PDC into CS1 and CS2, there is a tendency to think that the subjects are too advanced, or that they require too much infrastructure. However, because PDC has become so fundamental to practice, all modern programming languages include support for PDC, either through standard libraries or language constructs. These can be employed with cognitive loads similar to using common object classes, such as vectors, lists, sets, maps, and dictionaries.

It is important to understand that, in early courses, the goal in experiencing PDC is to open the student's mind to an appreciation that algorithmic problem solving can encompass concepts of parallelism and distribution. There is no expectation that students will gain a higher level of skill in implementing these concepts than they do with any other language features or classes employed at this level.

Another thought that often arises among instructors is that there is no space in these courses to add material. But our analysis of widely used textbooks and syllabi from select institutions showed that there is a great deal of material that is dealing with limitations arising from teaching around a computing model that dates to the 1980s.

For example, in CS1, it is typical to see branching and looping focused on parsing and error checking of text typed on a console or read from a text file, that is being stored into primitive types. Formatting console text output (e.g., printing tables with columns and headings) is another common pattern.

But modern user I/O APIs include data checking, and file I/O APIs enable writing and reading serialized objects, even with remote repositories (illustrating distribution of data). They thus avoid the impression that I/O is entirely about streams of chars on the local machine, which is an entirely foreign paradigm to internet-native students whose prior experience of computing is via graphical interfaces. Such interfaces also exemplify event-driven processing, which illustrates many basic ideas behind control parallelism.

In existing CS2 courses, considerable effort may be spent on implementing linked structures from primitive types, including building APIs that exhibit weak robustness, and using fixed algorithms where the efficiency doesn't scale with data volume. Modern container and collection classes can instead serve as examples of good design and provide opportunities to explore their underlying algorithms. Thus, there are indeed possibilities for making space to add PDC in these courses, when they are updated to use more modern tools.

What will also be seen in these chapters is that, in many cases, PDC concepts can be overlaid on existing topics with a modest investment of time. For example, if a recursive mergesort is already taught, it is easy to see that parallelism can be exploited by making the first few levels of calls spawn separate threads. Another approach is to introduce concepts of parallelism or distribution through unplugged activities, and then have students relate those to provided or scaffolded code that enables them to see the benefits of applying the concepts.

The development teams thus created examples of courses where PDC concepts related

to data parallelism, distributed data access, and event-driven processing are incorporated as a natural aspect of learning to solve problems algorithmically, which they then tested with their own students before presenting them to the testing teams for adoption.

0.3.2 Testing Team Recruitment

A primary component of this project was to ensure that course exemplars were tested and adopted in different contexts. Toward that end, we successfully recruited six testing teams, with institutions ranging from a community college to an MSI to an R1 university, both public and private, and geographically widely dispersed. We recruited teams using our CDER center mailing list, and listserves such as TCPP-announce and SIGCSE, plus promotion at our Edu* workshops, as well as our CDER booth at SIGCSE'24. We also held a webinar for the potential recruits, laying out the project goals and significance, application requirements, and expected outcomes and support available.

This groundwork resulted in good quality applications which were evaluated first on the basis of whether the applicant was teaching one of the target courses, whether their institution was open to modifying their CS1 and CS2 courses, and whether they were using one of the two target languages (Java or C++). A secondary basis for the evaluation was to balance the number of testing teams for each language and to ensure that the set of teams included a variety of institution types and sizes. We selected six teams comprising one or two faculty per team as described in Section 0.2.

It turned out that each of the teams had at least one person who had participated in one of our earlier training workshops. We held a kickoff meeting in May of that year laying out the project goals, timelines, and testing team responsibilities. Because the earlier training had included the IRB application process and evaluation methods, it was only necessary to review these. The development teams also provided a status report on their course development efforts. This was followed by a series of biweekly meetings where the teams explained their institutional contexts, described introductory course syllabi, and worked on plans for IRB approvals.

These early meetings led to an in-person summer workshop where the project personnel assembled. Because the testing teams all had prior training (and some had already developed and tested PDC course interventions as a result), they were able to hit the ground running, and the focus quickly shifted to brainstorming pedagogy and evaluation methodology.

0.3.3 Collaboration among Development and Testing teams, and support of Backbone team

Following the first summer meeting, the teams met regularly online. The CDER backbone team met weekly to plan the meetings with the other teams, and also to work on related activities by the center, such as our series of PDC education workshops associated with SC, IPDPS, and HiPC, tutorials and a booth at SIGCSE, development of a new monograph presenting examples of infusing PDC into courses, a journal special issue, planning the summer project meeting, and recruiting new trainees for a separate summer workshop.

The backbone and development teams met weekly to work on course development, review reports from the testing teams, and plan our meetings with them. All three teams met bi-

weekly, starting in the second year, as the testing teams were working on their IRBs and implementations. In the third year efforts shifted to running the new courses and gathering data. Many of the team members would get together at conferences to discuss the project, and additional summer meetings were held at the end of the second and third years.

0.3.4 Exemplar Creation and Testing by the Development Teams

0.3.4.1 TTU CS1 and CS2 Exemplar Development (C++)

Tennessee Tech students begin the semester with a basic understanding of sequential vs. parallel computing through an *unplugged penny sorting* group activity. The activity has several timed phases to show speedup with parallel phases as well as load balancing.

Students are introduced to topics related to distributed computing in the functions module. They are taught when a problem solution requires interaction with a *remote service* and they have a lab programming assignment where they pull and display data from a real-world, live, remote service.

Students are introduced to the range-based for loop (for-each loop) in the arrays/vectors module. Although it doesn't execute in parallel, the for-each construct exemplified dependence-free processing of a collection, which can be seen as a candidate for parallelism. They are also given an introduction to *data parallelism* and are taught how to identify algorithmic solutions that exploit it. Students have a lab assignment where they demonstrate data parallelism with a vector. Instruction has been modernized in this module to spend more time discussing the Vectors and Arrays classes in the C++ standard template library. The lab assignment is paired with an unplugged activity focusing on data parallelism. This unplugged activity is a *flag-maker activity* where students use markers to color in pixels of the flag in certain orders to demonstrate the use of for-each loops. This activity has several phases to demonstrate the following concepts: sequential vs. parallel programming, speedup, scalability, synchronization, underutilized resources, load balancing, pipelining, contention, and race conditions.

Near the end of the semester when object-oriented programming with classes and objects are introduced, students are introduced to *event handling* and taught when an algorithmic solution can benefit from the use of multiple threads.

For CS2, TTU modified a module where the performance of sorting algorithms was compared by including the use of *OpenMP* to parallelize a bubble sort (as an odd-even transposition sort), and a recursive merge sort. Because OpenMP makes this relatively easy, via the insertion of a few pragmas, it adds little to the time or cognitive load for the module, but enables students to clearly see the benefit of parallelism for sorting large vectors.

0.3.4.2 Knox CS1 and CS2 Exemplar Development (Java)

Knox College developed several modules for CS1. *Flag Maker:* Knox updated a version of an existing programming assignment in which students draw flags pixel by pixel using nested loops. It was changed to use JavaFX (previously in Swing) and to add clicker questions about the order in which pixel colors are assigned and which operations are independent. (The simplest way to draw some flags is in layers, which introduces dependencies if the drawing

is done in parallel.) An unplugged activity was also added (students using markers) to this module.

Testing "mystery code": A new tool for testing that lets students just give input/output pairs rather than explicitly using JUnit or similar frameworks. The assignment gives a series of methods and the students must figure out what they do by giving inputs and observing the outputs. This activity teaches lessons about confirmation bias, using distinct test cases, and verifying hypotheses with test cases. It is motivated as a shift toward testing and understanding code rather than writing it from scratch since generative AI can assist with writing.

Events and using a framework: Using the Greenfoot IDE as a framework for interactive code; code is attached to sprites and called by events from the system rather than from a main method. Students are introduced to the event-driven programming model and provided some examples, after which they created a variety of games (open-ended assignments).

Interactive server: Knox created several versions of this lab, which has students interact with a provided server to illustrate distributed data. The original version stored calendar dates and was not sufficiently interesting. For the next offering, students ran instructions to create 3D objects using a generalization of Logo (turtle graphics) to a Minecraft world. A version of r/place was also created, where participants can place pixels at a throttled rate to collaboratively create an image.

For CS2, Knox created a module introducing the *fork-join framework* for parallel computing. This gives students practice writing recursive code. Analysis is done in terms of work and span following the treatment of CLRS in a way that feels similar to the rest of the course. For programming assignments, students parallelize a graphical heat diffusion simulation (producing a visibly-faster program) and also implement a variety of parallel reduction operations.

0.3.5 Adaptation and Adoption by Testing Teams

At our first summer meeting, the CS1 exemplars from the development teams were presented, and then collaboration with the testing teams led to significant refinements to improve transferability and incorporate additional implementation ideas. During the following year, testing teams applied for IRB approval, and specific research questions were developed for each institution's study, with a focus on coordinated data collection for cross-team analysis. At the second summer meeting, data from the baseline runs, and some initial experimental CS1 runs by the testing teams was presented and then the focus turned to similar collaboration on the CS2 exemplars that had been developed, which were then tested in year three. The chapters in this book illustrate the outcomes of this process, where teams report how they adapted the exemplars to their unique contexts and existing course syllabi.

0.3.6 Evaluation Methods

We developed the following instruments that were administered during the project.

Student Driven:

1. Pre-course and post-course surveys assessing experience and attitudes toward PDC given to students by the development and testing team instructors.
2. Lab assignment and programming assignment surveys for students to determine how much time the students spend on the assignments, which days they worked on the assignments, what they learned from the assignment, the most challenging aspect of the assignment, if they still have questions about the assignment, and the extent to which they enjoyed the assignment.

Instructor Driven:

1. Pre-selection surveys with an objective component regarding the test team faculty applicants' background in PDC, current use of PDC in classes, institution, courses, student population, and a subjective component assessing initial attitudes toward incorporation of PDC into lower level classes.
2. Post-selection surveys of the Test Team faculty instructors on the appropriateness and effectiveness of the exemplars, likelihood of incorporation of PDC into other related classes, and initial ideas for the approaches and intervention points for incorporation of the exemplars in their existing courses.
3. Annual surveys (conducted during summer working sessions) of the Test Team faculty instructors to assess the number of students actually taught, degree to which PDC is incorporated, and the degree to which the interventions and evaluations are effective.
4. Post-adoption survey of the Test Team and Development Team faculty instructors to assess likelihood that the interventions will be retained, whether they will be reduced or expanded, and new directions being taken as a result of the experience.

0.4 Faculty Learning Community Formation

Community building: The project offered the participating faculty a sustained opportunity to integrate PDC content in their courses. Their engagement in this faculty learning community (FLC) allowed fruitful multi-institutional collaborations for developing and teaching the content. We can envision the faculty as change agents who amplified the impact of what they learned into widespread knowledge and use impacting a) the multiple cohorts of students who emerged with a better understanding of PDC concepts and b) the colleagues at their colleges and departments who wish to modernize the CS curriculum.

Sustainability: The project created exemplars for the introductory computing course sequence that will enable a broad set of programs to incorporate PDC concepts, and move forward to a 21st century computing paradigm. Our outreach to a set of testing teams set the stage for a sustainable and scalable effort to train a diverse set of instructors at diverse institutions in the benefits of incorporating PDC concepts from the beginning of a student's education in computing. Those testing teams, along with the teams developed the exemplar course sequence and the rest of the group participating in the project, now serve as a trained cohort of educators who can reach out to their colleagues at other institutions, eventually training many thousands of students in introductory computing courses per year. The course

exemplars developed in the project are now available online, via this book and the associated repository, so that adoption can continue.

0.5 Roadmap for Subsequent Chapters

This volume is designed for selective, adoption-oriented reading. Readers do not need to read every chapter before beginning to identify useful material. Instead, they should begin with their adoption goal, use Chapter 1 for CS1 or Chapter 10 for CS2 to understand the common activity families, and then consult one or more institutional chapters for local implementation details, teaching experience, evidence, and repository resources. The table of contents gives the full chapter sequence; this section provides a shorter guide to what to read first.

0.5.1 Start from Your Adoption Goal

Potential adopters usually approach this volume with different levels of commitment and different time constraints. Some may want one class activity that introduces PDC concepts without changing the course structure. Others may want a short module, a laboratory activity, or a more systematic infusion into CS1, CS2, or both courses. A useful first decision is therefore which part of the introductory sequence is being considered. Chapter 1 provides the common activity entry point for CS1, Chapter 10 serves the corresponding role for CS2, and readers interested in coordinated sequence-level adoption can use both common chapters together with paired institutional exemplars.

Figure 2 summarizes these entry points and three common scales of adoption.

For a single class activity, begin with Chapter 1 for CS1 or Chapter 10 for CS2 and look for an activity that fits the local course topic and available time. For a short module or lab, read the common description in the appropriate common chapter and compare two or three institutional chapters with similar constraints. For a course-level infusion, browse through Chapter 1 or Chapter 10 and scan the descriptor elements and course-change sections in the institutional chapters, then read the chapters that most closely match the local course language, class size, schedule, and instructional goals. For coordinated CS1–CS2 adoption, the paired institutional exemplars illustrate how introductory conceptual exposure can progress toward deeper implementation, correctness reasoning, synchronization, and performance analysis.

A useful first-adoption strategy is to choose the simplest version of an activity that fits local constraints. After one offering, instructors can expand the activity, add a plugged component, collect more systematic evidence, or connect the activity to a broader course- or sequence-level infusion.

0.5.2 CS1 and CS2 Exemplars at a Glance

Table 2 gives a compact view of the institutional CS1 and CS2 exemplars in this volume. Where both courses are represented, the CS1 and CS2 rows are grouped by institution to help readers see both the local context and the progression across the introductory sequence.

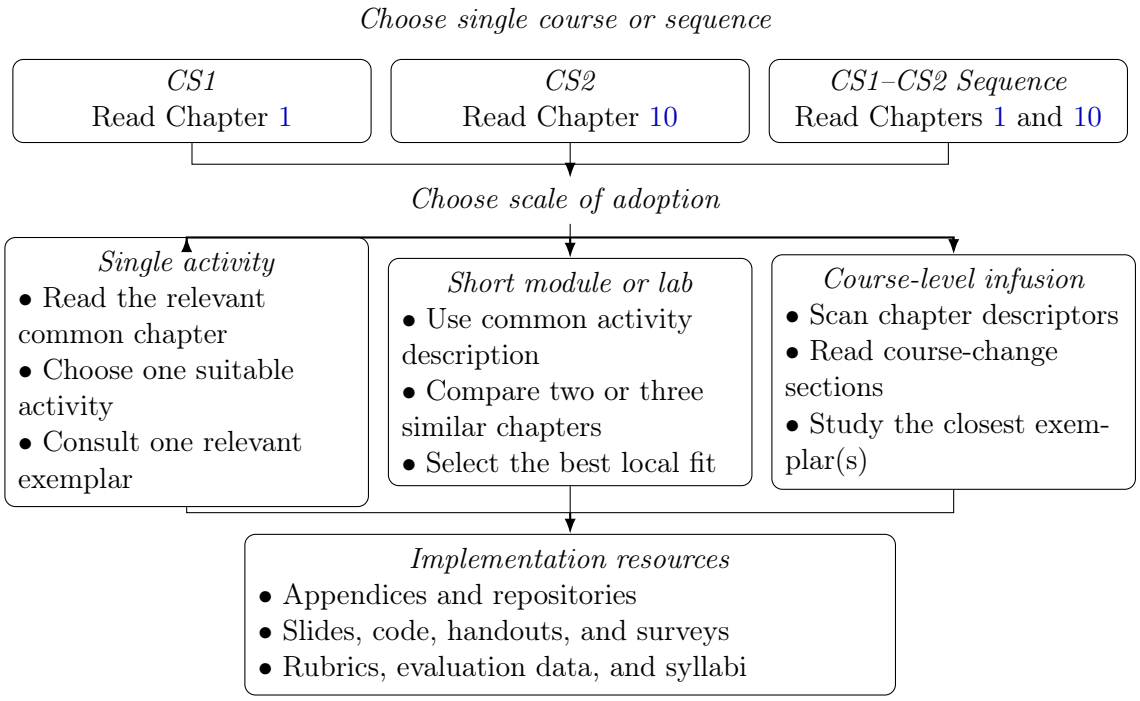


Figure 2: Suggested reading paths by course and scale of adoption.

Table 2: Quick comparison of CS1 and CS2 institutional exemplars for potential adopters.

Chapter	Institutional context	Language	Primary PDC infusion style	Good starting point for
TNTECH	**Mid-sized, public university**			
Ch. 2 (CS1)	Large CS1 lecture/lab setting	C++	Unplugged Penny and Flag Maker activities; OpenMP/data-parallel programming; remote-data/API assignment	Large-section CS1 adoption; programming-linked PDC infusion; multi-semester evidence
Ch. 11 (CS2)	Large Data Structures and Algorithms lecture/lab course with multiple sections	C++	OpenMP integrated with sorting and algorithm analysis; sequential/parallel comparison, timing, speedup, and scalability	Large-section CS2 adoption; parallel sorting and performance analysis
Knox	**Small residential liberal arts college**			
Ch. 3 (CS1)	Java-based introductory sequence	Java	Conceptual and programming-oriented CS1 activities adapted to a small-college context	Liberal-arts setting; smaller classes; Java-based adaptation
Ch. 12 (CS2)	Java-based CS2/Data Structures course	Java	Fork–Join framework integrated with recursion and algorithm analysis; work/span, dependency graphs, reductions, and race conditions	Small-class Java CS2; task parallelism through recursion and divide-and-conquer
USI	**Public master's-level institution**			
Ch. 4 (CS1)	Introductory object-oriented CS1 course	Java	Unplugged Flag Maker and Penny activities; plugged-in Flag Maker, Parallel Sort Penny, and Greenfoot; activity-level assessment	Structured activity adoption; evidence-rich unplugged and plugged-in examples

Table 2 continued.

Chapter	Institutional context	Language	Primary PDC infusion style	Good starting point for
Ch. 13 (CS2)	Intermediate object-oriented programming course	Java	Event-driven unplugged/plugged activities; Fork–Join decomposition and parallel sum; multithreaded parallel image processing	Structured multi-activity CS2 infusion; progression from conceptual activities to parallel implementation and performance analysis
UNL	**Large R1 institution**			
Ch. 5 (CS1)	Multiple CS1 sections and computing majors	C, Java	Codeless modules, visual/conceptual activities, remote-data activities; low/no-code pathway; large-institution adaptation	Large-enrollment, online/asynchronous/hybrid courses; low-code entry introduction
Ch. 14 (CS2)	Java-based CS2 course; selected existing labs modified	Java, JDBC, SQL	Parallel merge-sort benchmarking and connection pooling in a multithreaded application	Minimal-change CS2 adoption through existing sorting and database-oriented labs
Webster	**Private primarily undergraduate institution**			
Ch. 6 (CS1)	Small-to-moderate CS1 sections	C++	Animations, visualizations, performance examples, and game/simulation-oriented activities	Visualization-first adoption; low-barrier conceptual entry points
Ch. 15 (CS2)	C++ CS2 course emphasizing recursion, sorting, data structures, and algorithmic problem solving	C++	Serial and OpenMP parallel merge sort; task decomposition, timing, parallel overhead, and performance interpretation	C++ CS2 adoption with limited curricular disruption; sorting- and performance-centered parallelism
Ch. 16 (CS2)	Python-based CS2/Data Structures pathway	Python	Threading and multiprocessing; parallel image processing with OpenCV; speedup, efficiency, scaling, overhead, and memory effects	Python CS2 adoption; multiprocessing, image processing, and experimental performance analysis
Casper	**Two-year community college**			
Ch. 7 (CS1)	Small cohorts and mixed student pathways	C++, Python	Custom digital-textbook material, unplugged activities, OpenMP, remote data, and physical computing	Community-college setting; small cohorts; varied delivery formats
Ch. 17 (CS2)	CS2 course with small cohorts and varied prior PDC exposure	C++, Java	Unplugged Penny, Coin Addition, and Fork–Join activities; OpenMP array/sorting exercises; Greenfoot project	Community-college CS2; conceptual plus programming activities; useful with limited prior CS1 PDC exposure
HPU	**Private undergraduate institution**			
Ch. 8 (CS1)	Small class sizes	Java	Flag Maker, Penny Sorting, lightweight plugged/code-review extension, and low-resource activities	Small-scale first adoption; low-preparation unplugged activities
MSU	**Public university**			
Ch. 9 (CS1)	Focused CS1 intervention	Java	Flag Maker-centered data-parallelism activity with supporting animations and assessment	Minimal-infusion model; short, focused CS1 intervention
Ch. 18 (CS2)	Java CS2 course continuing the CS1 sequence	Java	Event-driven Greenfoot game and Java multithreaded image processing; data parallelism, synchronization, speedup, and Amdahl's Law	Focused two-module Java CS2 infusion; progression from conceptual CS1 exposure to multithreaded implementation

Readers should treat the table as a starting filter. The most important adoption questions are not only "Which language matches mine?" but also "Which activity scale, preparation level, evidence type, and implementation style match my course?"

0.5.3 Using the Common Chapters with Institutional Exemplars

Chapter 1 and Chapter 10 provide the common entry points for CS1 and CS2, respectively. These chapters distill recurring activity families, shared PDC concepts, common implementation challenges, and cross-institution lessons learned. The common chapters are organized around reusable activities and cross-institution guidance, whereas the institutional chapters show how those activities and concepts were incorporated into actual courses. For CS1, Chapter 1 provides activities such as Flag Maker, Penny Search or Penny Sorting, animations and visualizations, remote-data/API work, event-driven examples, and lightweight programming assignments. For CS2, Chapter 10 emphasizes event-driven programming, Fork–Join and task parallelism, parallel sorting, parallel image processing, shared-memory programming, and performance analysis. Figure 3 shows how the two common chapters are organized.

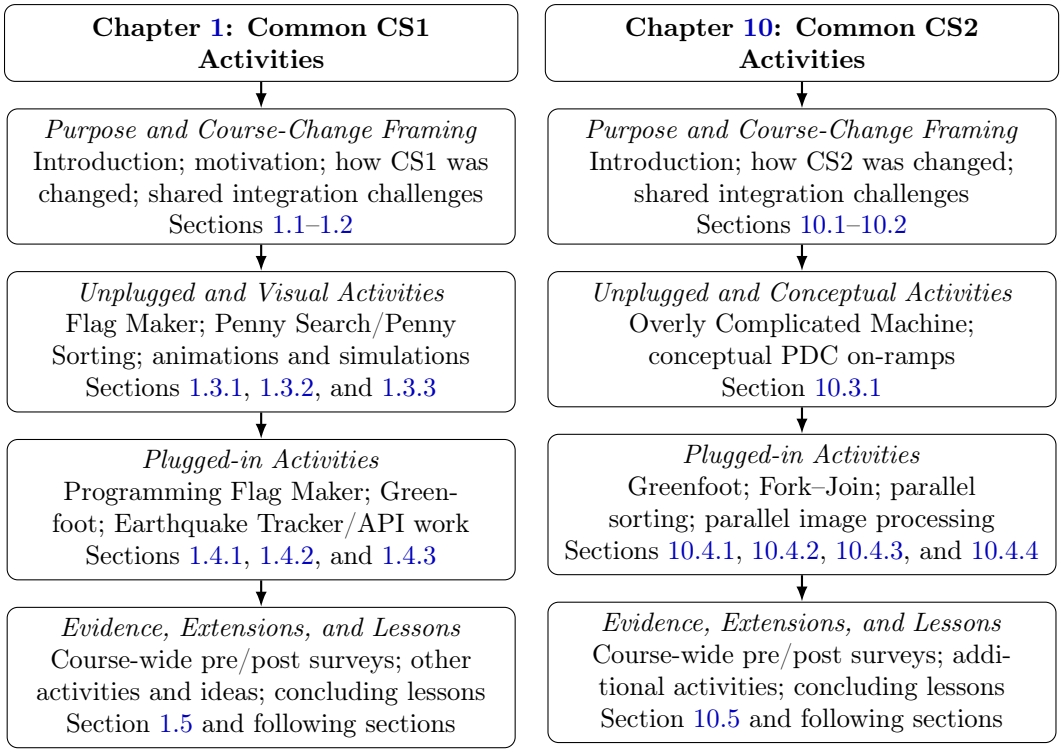


Figure 3: Organization of Chapters 1 and 10 – common CS1 and CS2 activities and adoption guidance.

Common Institutional Chapter Structure

The institutional chapters follow a common structure so that readers can compare exemplars without first learning a new organization for each chapter. As summarized in Fig-

ure 4, each chapter moves from quick screening information, to implementation details, to adoption-ready appendix and repository resources.

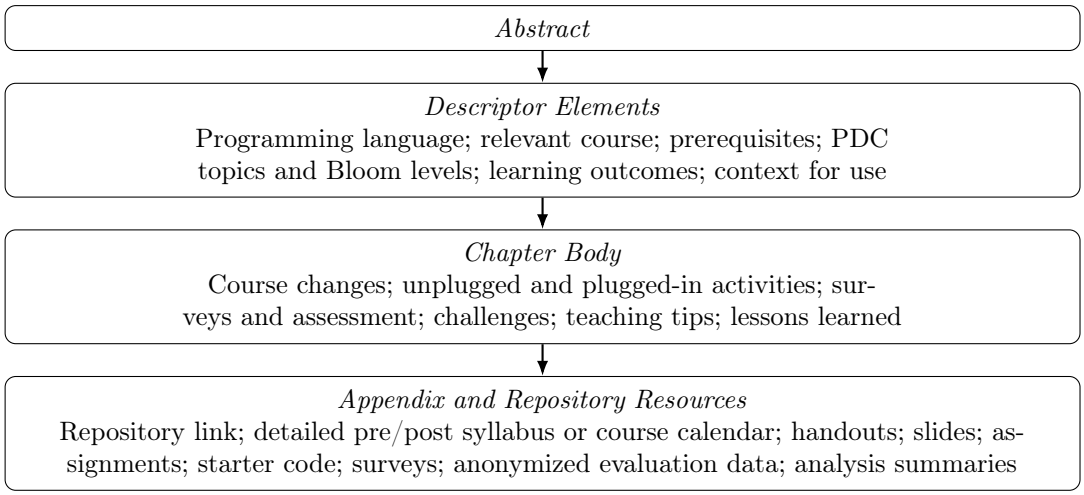


Figure 4: Common structure of institutional exemplar chapters.

The abstract and descriptor elements help adopters quickly decide whether a chapter matches their course setting, language, prerequisites, instructional goals, and desired PDC topics. The chapter body then explains how the course was changed and presents the major activities, including classroom logistics, teaching guidance, challenges, and available evidence. The appendix and repository resources provide the concrete materials needed for adoption, such as syllabi, course calendars, handouts, slides, assignments, starter code, surveys, anonymized data, and analysis summaries. Where available, these materials are organized around four recurring components: a repository link, detailed pre/post syllabus or course-calendar material, organization of the instructional resources, and survey/evaluation data and analysis.

0.5.4 Finding Materials Quickly

The fastest way to use this volume is to combine chapter-level navigation with search. The table of contents identifies the major chapters and sections. The abstract and descriptor elements provide quick overview of each institutional chapter. For activity-oriented browsing, Chapters 1 and 10 provide common CS1 and CS2 activity families and pointers to institutional implementations. The PDF search can be used to locate activity names, PDC concepts, tools, and evidence terms.

Some useful search terms include: Flag Maker, Penny Search, Penny Sorting, OpenMP, Earthquake Tracker, Greenfoot, animation, visualization, data parallelism, speedup, load balance, event handling, remote data, and API, as well as Fork-Join, work and span, race condition, synchronization, parallel sorting, parallel image processing, benchmarking, and scalability. Readers looking for assessment material should search for terms such as pre/post survey, student feedback, engagement, assessment, reflection, and data analysis.

0.5.5 Summary Guidance and Conclusions

This volume is best used as a guided collection of adaptable exemplars rather than as a linear book. Begin with an adoption goal, use Chapter 1 for CS1 or Chapter 10 for CS2 to identify relevant activity families and common adoption guidance, compare a small number of institutional chapters that resemble the local context, and then move to appendices and repositories for implementation resources. A small, feasible first adoption is often the best starting point for broader course or introductory-sequence modernization.

Taken together, the project exemplars show that parallel and distributed computing concepts can be introduced in CS1 and CS2 without wholesale redesign of the introductory curriculum. The project produced 16 institutional course exemplars across eight institutions, two common activity-and-lessons chapters, and reusable collections of syllabi, classroom activities, assignments, code, surveys, and evaluation materials that can continue to be refined and shared through the CDER repository. These outcomes build on a longer CDER effort: the TCPP Curriculum Initiative helped define what undergraduates should learn about PDC; early-adopter programs and instructor training demonstrated both the promise and challenges of incorporating these concepts into real courses; the CDER book series and courseware repository provided mechanisms for dissemination; and NSF-supported projects enabled the development, transfer, and evaluation of increasingly complete instructional models (see more in the Appendix). This e-book represents the next step in that progression—from guidelines and individual modules toward classroom-tested introductory course exemplars that instructors and departments can adapt to their own settings.

Our broader vision is to make emerging computing concepts a routine part of the early undergraduate curriculum rather than topics deferred to advanced courses. Building on this work, a new ACAI-Train project will extend the exemplar-based approach through scalable instructor training for integrating AI and big data concepts into CS1, CS2, and Systems courses. A central mechanism will be annual in-person instructor-training workshops, supplemented by hybrid and online training opportunities, through which instructors can learn, adapt, implement, and evaluate these materials in their own courses. This continuing cycle of exemplar development, instructor preparation, classroom implementation, and shared experience is intended to build a broader community capable of sustaining curriculum modernization.

We invite instructors to use and adapt these resources, contribute their implementation experiences and updated resources to the CDER repository for broader dissemination, participate in future training and other CDER activities, and become ambassadors for modernizing the introductory computing curriculum.

Appendix

Prior Work Framing this Project

The NSF-supported Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER)[4] has worked for more than a decade to integrate suitable parallel and distributed computing concepts into undergraduate computing curricula, with particular emphasis on early and core courses with a premise that all computing graduates have a certain skill set in PDC for the modern workforce. The CS1/CS2 exemplar project described in this volume did not arise in isolation. It builds on a sequence of curriculum-development, early-adoption, instructor-training, dissemination, and community-building efforts that together shaped the need for this project and made it feasible. In particular, the project draws on the IEEE TCPP Curriculum Initiative, CDER early adopter programs, the CDER book series, NSF-supported conceptualization and implementation grants, the Edu* workshop series, and training workshops for instructors.

The TCPP Curriculum Initiative: The IEEE Computer Society Technical Committee on Parallel Processing (TCPP) Curriculum Initiative began with a central question in 2010 during which Prasad was its chair: what should every computer science and computer engineering undergraduate know about parallel and distributed computing? The resulting curriculum guidelines identified PDC topics across programming, architecture, algorithms, and cross-cutting areas, and mapped these topics to possible levels of coverage and courses in the undergraduate curriculum. Version I of the TCPP curriculum was released in 2012 [51], and Version IIβ was released in 2020 with expanded attention to Big Data, energy, distributed computing, and pervasive topics [50].[5] The initiative helped make the case that PDC is not only an advanced elective topic, but a body of concepts that should be introduced throughout the undergraduate curriculum, including in early core courses. The ACM/IEEE Computer Science Curricula 2013 report explicitly linked to the TCPP curriculum for more complete PDC coverage, and CS2023 includes a dedicated PDC knowledge area that cites the Version IIβ effort. This national curriculum framing provided the intellectual foundation for moving from topic guidelines toward concrete course exemplars.

Early Adopters: Curriculum guidelines alone do not change courses; instructors need examples, incentives, community, and evidence. The CDER early adopter competitions were created to incentivize instructors who were willing to incorporate PDC topics into existing undergraduate courses and report on their experiences. Early adopter awards funded by NSF, TCPP, Intel, NVIDIA, and IBM provided modest seed funding and some equipment, but their broader purpose was to create a community of practitioners who could test, adapt, and refine PDC curricular ideas in varied institutional settings. Over multiple competitions, more than 140 early adopters were supported, and the broader CDER community of early adopters and trained instructors has grown to roughly 300 participants or institutions nationally and internationally. These early adopters demonstrated that PDC could be infused into many parts of the curriculum, but they also revealed a persistent need: instructors wanted more

[4] https://cdercenter.org/pdc-center/
[5] https://cdercenter.org/pdc-curriculum/

complete, coherent, classroom-tested course exemplars, especially for early courses where adoption barriers are highest.

CDER Book Series: The CDER book series provided another mechanism for collecting, curating, and disseminating PDC teaching materials. The first two volumes, published by Morgan Kaufmann and Springer, gathered instructor- and student-facing chapters on topics such as introductory parallelism, threads, fork-join parallelism, performance, scalability, energy efficiency, MapReduce, GPU computing, mobile concurrency, and integrative GUI applications. These volumes have been made available as free preprint chapters through the CDER website and have collectively reached tens of thousands of chapter downloads. A third CDER volume – expected to be published in late 2026 by Springer – focused on adopter experience, exemplars, and resources, extends this model by emphasizing implementation context and transferability. The present CS1/CS2 exemplar e-book follows the same broad philosophy, but focuses more tightly on modernizing the introductory course sequence through complete course packages, shared activities, assessment instruments, and institutional adaptation narratives, with a vision of creating national models.

Grants and Support: Several NSF-supported efforts directly frame this project. The earlier CDER center grant supported the development of PDC curriculum guidelines, dissemination, community building, and early adoption. A subsequent NSF CyberTraining implementation grant supported broader instructor training for incorporating PDC into undergraduate CS and CE curricula. A CyberTraining Institute Conceptualization grant supported stakeholder workshops that examined the systemic barriers to broader PDC adoption, including the continued dominance of a sequential model of computation in early courses, the lack of modern course exemplars, limited instructor preparation, and the difficulty of changing tightly coupled curricular structures. The resulting message was clear: many stakeholders agreed that computing education needed modernization, but potential adopters needed someone to "go first" and show what such modernization could look like in real courses. The NSF Modernizing CS1/CS2 Exemplar Project, Awards #2321015-20, directly addresses that need by developing, transferring, and evaluating PDC-infused introductory course exemplars across diverse institutions.

Edu* Workshops and Community Building: The Edu* workshop series has been a central mechanism for building and sustaining a community of PDC educators. EduPar, held with IPDPS, began as a venue for early adopters and other interested instructors to report experiences, receive feedback, and discuss curricular and pedagogical issues in parallel and distributed computing education. EduHPC, held with SC, created a complementary venue connected to the high-performance computing community, while EduHiPC, held with HiPC in India, extended the model internationally. Together, EduPar, EduHPC, and EduHiPC have provided recurring forums for papers, posters, panels, working meetings, special issues, and community feedback. These workshops have helped transform isolated adoption efforts into an ongoing professional community of instructors, curriculum developers, and PDC researchers interested in undergraduate education.

Instructor Training and Related Activities: Distinct from the Edu* research and community workshops, CDER has also organized instructor-facing training activities aimed at helping faculty prepare, adapt, implement, and evaluate PDC-infused course materials. These include NSF CyberTraining week-long summer workshops, shorter tutorials and special sessions at SIGCSE, training activities associated with EduHiPC, and hybrid or online training events. Such workshops emphasize hands-on preparation: introducing PDC concepts, walking through classroom activities and programming assignments, helping instructors identify appropriate course insertion points, and supporting assessment plans. This training infrastructure helped prepare early adopters and later this exemplar-project participants to move from interest in PDC education to concrete classroom implementation. All our development and testing team members were past trainees or early adopters.

Acknowledgements

This work was supported in part by the National Science Foundation through Award #2321015-20, *Modern Course Exemplars Infused with Parallel and Distributed Computing for the Introductory Computing Course Sequence*, and builds on related CDER Center efforts supported by earlier NSF awards, including NSF CyberTraining Institute Conceptualization Award #2002649, and NSF CyberTraining Broadening Adoption Award, #2017590. We gratefully acknowledge the contributions of the development teams, testing teams, backbone team, participating postdocs and students, teaching assistants, workshop participants, and CDER community members whose course designs, classroom implementations, evaluation efforts, and feedback shaped this volume. All substantive ideas, instructional designs, analyses, interpretations, and conclusions presented in this e-book are those of the chapter authors and project team.

Generative AI tools, including large language models, were used selectively to assist with prose polishing, Overleaf/LaTeX formatting of selected figures and tables, debugging compilation issues, cover-page styling, and editorial checking. The authors reviewed and are responsible for the final content.

PART I - Computer Science 1 (CS1)

Chapter 1

Common CS1 Parallel and Distributed Computing Activities and Lessons

***Shared Activity Families, Adoption Guidance, and Evidence*†**

April R. Crockett (Tennessee Technological University),
Srishti Srivastava (University of Southern Indiana),
David P. Bunde (Knox College),
Xiaoyuan Suo (Webster University),
Charlotte Gruner (Casper College),
Jaime Spacco (Knox College),
Mary L. Smith (Hawai'i Pacific University),
Jiayin Wang (Montclair State University),
Chris Bourke (University of Nebraska–Lincoln),
Michelle Zhu (Kennesaw State University),
Peter Maher (Webster University),
Gerald C. Gannod (Tennessee Technological University),
Alan Sussman (University of Maryland, College Park),
Neena Thota (University of Massachusetts Amherst),
Charles Weems (University of Massachusetts Amherst),
Ramachandran Vaidyanathan (Louisiana State University), and
Sushil K. Prasad (University of Texas at San Antonio)

†**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 31
1.1 Introduction . . . 32
1.2 How CS1 was changed and PDC topics integrated . . . 34
1.2.1 Challenges . . . 35
1.3 New Unplugged Activities . . . 36
1.3.1 Flag Maker . . . 36
1.3.1.1 Activity Summary . . . 36
1.3.1.2 Prerequisites . . . 37
1.3.1.3 Learning Outcomes . . . 37
1.3.1.4 Logistics . . . 37
1.3.1.5 Running the Activity . . . 38
1.3.1.6 Activity Discussion . . . 39
1.3.1.7 Instructional Material . . . 39
1.3.1.8 Example Implementations . . . 40
1.3.1.9 Experience and Teaching Tips . . . 40
1.3.1.10 Data and Analysis . . . 40
1.3.2 Unplugged Penny Search Activity . . . 40
1.3.2.1 Activity Summary . . . 40
1.3.2.2 Prerequisites . . . 42
1.3.2.3 Learning Outcomes . . . 42
1.3.2.4 Implementation Details . . . 42
1.3.2.5 Experience and Teaching Tips . . . 44
1.3.2.6 Data and Analysis . . . 45
1.3.3 Using Animations . . . 47
1.3.3.1 Parallel Search Animation . . . 47
1.3.3.2 Parallel Linked List Animation . . . 47
1.3.3.3 Flag Coloring Animation . . . 48
1.3.3.4 Zombie Attack Animation . . . 49
1.3.3.5 Experience and Teaching Tips for all the Animations . . . 50
1.3.3.6 Data and Analysis . . . 51
1.3.3.7 Example Implementations . . . 52
1.4 New Plugged-in Activities . . . 52
1.4.1 Programming Flag Maker . . . 52
1.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 53
1.4.1.2 Implementation Details . . . 54
1.4.1.3 Experience and Teaching Tips . . . 56
1.4.1.4 Data and Analysis . . . 57
1.4.2 Greenfoot Activity . . . 57
1.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 57
1.4.2.2 Algorithms . . . 58
1.4.2.3 Implementation Details . . . 58
1.4.2.4 Experience and Teaching Tips . . . 60
1.4.2.5 Data and Analysis . . . 61
1.4.3 Plugged-In Earthquake Activity . . . 61
1.4.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 61
1.4.3.2 Algorithms . . . 62

1.4.3.3 Example Implementations 62
1.5 Course-wide Pre and Post Course Surveys 62
1.6 Conclusion 63

Chapter figures

1.1 Flag of Mauritius [65] 37
1.2 Graphical depiction of the Flag Maker scenarios 38
1.3 Modified ASPECT survey used for the Penny Sorting unplugged activity 46
1.4 Parallel Search Animation 47
1.5 Parallel Linked List Animation 48
1.6 Flag Coloring Animation 49
1.7 Zombie Attack Animation 50
1.8 Participant Responses from Teaching Animations 51
1.9 Participant Responses from Teaching Animations 52
1.10 The Lithuanian flag, as rendered by the Flag Maker assignment. 53
1.11 The draggable thumb at the bottom replays grid square placement. While not directly parallel, the thumb can nonetheless demonstrate parallelism. 54
1.12 The assignment piece for the Japanese flag. 55
1.13 Solution as text for the Lithuanian flag. 56
1.14 The Hedgehog example in Greenfoot. 58
1.15 How to move an actor in Greenfoot. 59
1.16 Example of setActOrder() method 59

Chapter tables

1.1 Common CS1 PDC activity finder 32
1.2 Additional institutional CS1 PDC activities 33

Abstract

This chapter distills some common CS1 activities, implementation strategies, challenges, and lessons learned across the eight institutional CS1 exemplars in this volume. It serves as a shared resource chapter between the project overview in Chapter 0 and the institution-specific CS1 chapters 2–9 that follow. Rather than presenting a local course implementation, the chapter identifies activity families employed across multiple institutions and reusable guidance that cut across TNTECH, Knox, USI, UNL, Webster, Casper, HPU, and MSU. Its purpose is to help instructors locate activities that can be adopted as a single class activity, a short module, a laboratory exercise, or part of a broader course-level infusion of parallel and distributed computing (PDC) into CS1, while ensuring that basic CS1 learning outcomes are preserved.

The activities summarized here introduce accessible versions of core PDC ideas, including data parallelism, task decomposition, speedup, load balance, contention, communication, distributed data access, APIs, event handling, asynchrony, and limits of parallelism. These ideas are connected to familiar CS1 topics such as loops, arrays, searching, sorting, functions, objects, graphical interaction, data access, and basic algorithmic reasoning. Recurring activity families include unplugged activities such as Flag Maker and Penny Search/Penny Sorting; visualizations and simulations for parallel search, linked lists, flag coloring, and Zombie Attack; and lightweight programming activities involving remote earthquake-data APIs, Greenfoot or event-driven programming, and parallel search or sorting.

The chapter points readers to institutional chapters and their repositories, appendices, assessment instruments, student feedback, and classroom evidence so they can select and adapt materials that match their own teaching context. The repositories will be periodically curated and updated with instructional material and evidence.

Across institutions, shared lessons include starting with intuition, using unplugged or visual activities before code, keeping programming tasks lightweight, and embedding PDC concepts within existing CS1 learning goals rather than treating them as disconnected advanced content.

1.1 Introduction

Chapter 0 presents the overall motivation, project design, and roadmap for this volume. This chapter now turns from the project-level view to the shared CS1 activities and lessons that emerged across the institutional exemplars. Unlike Chapters 2–9, which describe local course implementations, this chapter is organized around reusable activity families. Its purpose is to help instructors quickly identify activities that may fit their own CS1 course and then move to the institutional chapters for nuanced implementation details, adaptations, evidence, and repository resources.

The chapter is intended for selective, adoption-oriented reading. An instructor may use it to find a single unplugged activity, a short visualization, a programming assignment, an assessment instrument, or a starting point for broader CS1 infusion. The activities summarized here are deliberately connected to familiar CS1 topics. In unplugged activities, students may first act as processors, divide work, observe idle time, or predict the behavior of an animation before seeing how the same ideas connect to programming concepts such as loops, arrays, functions, objects, APIs, graphical interfaces, or real-world data. This conceptual-to-practical progression helps introduce PDC without requiring students to become parallel programmers in their first course.

Table 1.1 provides a quick activity finder for this chapter. The first column briefly describes each activity and points to the section where readers should begin. The next two columns highlight the PDC ideas and the CS1 connections they reinforce. The last column cites which institutions have employed these and their estimates of classroom time utilized.

Table 1.1: Activity finder for CS1 PDC activities described in this common chapter.

Activity and brief description	Primary PDC ideas	CS1 anchors	Duration / where used
Unplugged, visual, and simulation activities			
Flag Maker. Students act as processors coloring flag-grid cells in sequential and parallel scenarios. §1.3.1.	Data parallelism; task decomposition; speedup; contention; pipelining; critical path; race conditions.	Loops; grids; repeated operations; graphical output; independent work and dependencies.	40–75 minutes. See TNTECH §2.3.2, Knox §3.3.1, USI §4.3.1, HPU §8.3.1, and MSU §9.3.1.
Penny Search / Penny Sorting. Students search or sort pennies under sequential, balanced-parallel, and load-imbalanced scenarios. §1.3.2	Data parallelism; load balance/imbalance; slowest worker; communication overhead; Amdahl's Law; observed versus ideal speedup.	Searching/sorting; timing; comparing strategies; reasoning from measured performance.	Fits a 50- or 75-minute class. TNTECH: 30–40 minutes of a 75-minute lab §2.3.1; USI: 75 minutes §4.3.2; HPU: 45–65 minutes §8.3.2; Casper: §7.3.2.
Animations and simulations. Students use visualizations of parallel search, linked-list concurrency, flag coloring, and Zombie Attack behavior. §1.3.3.	Work partitioning; speedup limits; synchronization; race conditions; mutual exclusion; dependencies; domain decomposition.	Arrays/lists; linear search; linked-list operations where covered; grids; prediction before code.	Webster animations are 10–15 minutes each: Animation §1.3.3; Zombie Attack may introduce a longer assignment: Zombie Assignment §1.3.3.4.
Plugged-in programming activities			
Programming Flag Maker. Students draw simplified flags on a 2D grid using nested loops and graphical output. §1.4.1.	Independent drawing operations; dependency reasoning; data-parallel structure; possible grid decomposition.	Nested loops; row/column indexing; 2-D grids; graphical output; testing.	Knox: 70-minute lab, plus one week assignment, §3.3.1. USI: 75-minute launch plus one week paired programming §4.4.1.

Continued on next page.

Table 1.1 continued.

Activity and brief description	Primary PDC ideas	CS1 anchors	Duration / where used
Greenfoot / event-driven work. Students build or modify graphical programs whose objects respond to events. § 1.4.2.	Event handling; asynchronous responses; active-object interaction; framework-controlled execution.	Objects; methods; classes; conditionals; graphical interaction; game/simulation logic.	Knox uses 2 labs in event-driven/OOP portion of CS1, §3.3.3. USI: optional 75-minute demo §4.4.3.
Earthquake Tracker / remote data. Students retrieve USGS data, parse JSON, filter events, and display selected information. §1.4.3.	Distributed data access; client-server model; APIs; JSON data; retrieve–parse–filter–display workflow.	Functions; loops; arrays/collections; libraries; Makefiles; data processing; external input.	TNTECH: §2.4.1; Casper: §7.4.3; UNL: §5.6.1 (about 1.25 hours for USGS encapsulation lab; cf. Table 5.3).
Assessment and evidence			
Course-wide surveys. Common pre/post items capture background and self-reported experience with programming and selected PDC concepts. §1.5.	Parallel processing, distributed computing, event handling; pre/post comparison; adoption evidence.	Programming background; languages; data types; expressions; branching; loops; functions; arrays; classes/objects.	Usually administered near course start and end.

A useful reading strategy is to begin with the activity family that best matches the local adoption goal. An instructor seeking a single class activity may start with Flag Maker, Penny Search/Penny Sorting, or one of the animations. An instructor seeking a short lab or programming assignment may begin with Programming Flag Maker, Greenfoot, or Earthquake Tracker. An instructor planning broader CS1 infusion should read Section 1.2 together with the institutional chapters to compare how different teams created space for PDC, aligned activities with existing CS1 topics, and collected evidence.

Additional CS1 PDC activities not described in this chapter are listed in Table 1.2.

Table 1.2: Additional institutional CS1 PDC activities not described as full activities in this common chapter.

Activity and brief description	Primary PDC ideas	CS1 anchors	Duration / where used
Unplugged, visual, or simulation activities			
zyBooks parallel-computing section. Students complete an interactive textbook section as a low-barrier conceptual introduction to parallel computing.	Parallel-computing vocabulary; conceptual exposure; preparation for later activities.	Interactive textbook work; vocabulary; preparation for programming examples.	Casper: §7.3.1. Estimated time for students to complete section is 10–15 minutes.
Unplugged Coin Addition. Students collaboratively add coin values to illustrate shared work, shared state, and synchronization issues.	Shared state; race conditions; synchronization; coordination.	Variables; accumulation; updating shared totals; order of operations.	Casper: §7.3.3. Estimated time for completion 30-40 minutes
MP Writing Activity. Students coordinate a writing task to experience communication, sequencing, and limits of parallel work.	Message passing; communication; coordination; limits of parallel work.	Instruction writing; sequencing; communication clarity.	HPU: §8.6.1; Table 8.12. USI: §4.6.1; Table 4.17.
Plugged-in activities and extensions			
Plugged-In OpenMP Data Parallelism Activity. Students use OpenMP to experiment with data-parallel work and observe speedup and setup overhead.	Shared-memory parallelism; threads; data parallelism; speedup; scheduling/setup overhead.	Arrays; loops; compilation; command-line execution; runtime interpretation.	TNTECH: §2.4.2; Figure 2.16 and related figures.

Continued on next page.

Table 1.2 continued.

Activity and brief description	Primary PDC ideas	CS1 anchors	Duration / where used
Plugged-In OpenMP Array Creation and Array Sum Activities. Students use OpenMP in array-creation and summation tasks to explore work sharing and parallel execution.	Shared-memory parallelism; threads; work sharing; shared variables; reduction-style reasoning.	Arrays; loops; variables; summation; compiler/setup practice.	Casper: §7.4.1, §7.4.2. Estimated time to completion 90 minutes.
Plugged-In Parallel Sort Penny Activity. Students use a plugged-in version of penny sorting to compare sequential and parallel strategies.	Parallel decomposition; sequential/parallel comparison; speedup; workload division.	Sorting; arrays/collections; code tracing/review; performance interpretation.	USI: §4.4.2; Table 4.12. HPU: §8.4.1; Table 8.8.
Codeless PDC modules. Students complete low-setup modules that introduce PDC concepts without requiring full parallel-programming implementation.	Parallel/distributed models; decomposition; speedup; PDC vocabulary; low-setup exposure.	Module completion; conceptual reasoning; short responses; asynchronous learning.	UNL: §5.4.1; Table 5.3.
Zombie Attack programming assignment. Students extend a grid-based simulation in which neighboring regions interact as infection spreads.	Domain decomposition; boundary interaction; state updates; neighboring-region communication.	2-D grids; simulation loops; conditionals; object/state updates.	Webster Zombie Attack Assignment section. Section §6.4.1
Knoxcraft. Students work in an interactive, game-like programming setting that connects objects, events, and framework-mediated behavior.	Event-driven interaction; object behavior; possible framework-mediated execution.	Objects; methods; game-style interaction; program structure.	Knox: §3.3.2. 70-minute lab, which turns into a week-long programming assignment.
Physical computing projects. Students use Raspberry Pi/GPIO projects involving traffic lights, buttons, multiple processes, and a temperature sensor.	Multiple processes; event-driven behavior; hardware interaction; concurrency-related reasoning.	Python; functions; hardware I/O; sensors/buttons; debugging.	Casper: §7.4.4, §7.4.5, §7.4.6. Time to completion varies, project 1 is approximately 50 minutes, project 2 is 90 minutes, project 3 is 4-5 hours to complete the 4 parts.

The remainder of the chapter follows this adopter-facing structure. Section 1.2 summarizes how CS1 was changed across institutions and the challenges that arose. Sections 1.3.1, 1.3.2, and 1.3.3 describe unplugged, visual, and simulation-based activities. Sections 1.4.1, 1.4.2, and 1.4.3 describe plugged-in activities. Section 1.5 summarizes the common course-wide pre/post survey structure. The conclusion distills cross-institution lessons for instructors who wish to adopt, adapt, or extend these materials.

1.2 How CS1 was changed and PDC topics integrated

Across the participating institutions, parallel and distributed computing topics were incorporated into CS1 through three broad, and sometimes overlapping, strategies: selectively reducing existing content, modifying current assignments and activities, and adding short PDC-focused modules to the established course [18]. The institutions represented a wide range of instructional settings, including public and private universities, small liberal arts colleges, and larger research institutions. Their courses also differed in programming language, class size, contact hours, laboratory structure, and semester length. Because of these differences, the project did not prescribe a single model for changing CS1. Instead, each institution adapted the materials to fit its local curriculum while preserving the foundational programming knowledge expected of beginning students.

Some institutions created space by reducing or deferring topics considered specialized,

redundant, or better suited for CS2. Examples included binary and random-access files, C-style strings and arrays, detailed output formatting, two-dimensional arrays, advanced class relationships, static members, garbage collection, or additional practice with object-oriented concepts. In most cases, these topics were not viewed as unimportant; rather, instructors determined that they could be covered more briefly or revisited in later courses. Other institutions retained the full CS1 syllabus and introduced PDC without removing content. One institution, for example, placed three asynchronous PDC modules outside regular class time, while another used existing laboratory and recitation periods to provide PDC activities without restructuring the lecture sequence. Across institutions, the amount of direct instructional time devoted to PDC ranged from no scheduled class time for asynchronous modules to approximately seven hours, generally representing one to two weeks of student engagement.

The most common strategy was to embed PDC concepts into topics already taught in CS1. Existing instruction on loops, arrays, searching, file processing, classes, objects, and problem decomposition was reframed to include ideas such as data parallelism, task decomposition, distributed data access, event handling, scalability, speedup, and computational cost. Examples included modifying array summation and searching assignments to discuss parallel execution, using serial and parallel versions of programs so students could compare execution times, accessing earthquake data through remote APIs, introducing OpenMP within an arrays laboratory, and using event-driven game programming to teach classes and objects. At other institutions, Flag Making §1.3.1, Penny Search §1.3.2, Minecraft §3.3.2, visualizations, animations §1.3.3, and other exercises were used to help students reason about parallelism before encountering implementation details. These changes generally layered PDC learning objectives onto traditional CS1 objectives rather than replacing them. Students still practiced loops, arrays, classes, file processing, and algorithmic thinking, but did so within contexts that reflected contemporary multicore, distributed, and event-driven computing.

The form of PDC instruction also varied according to institutional needs and instructor expertise. Some courses emphasized conceptual understanding through readings, discussions, animations, and codeless demonstrations. Others included guided programming exercises using OpenMP §2.4.2, Java parallel libraries, remote data sources, or event-driven environments such as Greenfoot §1.4.2. Several institutions used unplugged activities in which students physically modeled processors, divided work, or compared serial and parallel approaches. This variety was intentional: the goal was not to make students proficient parallel programmers during their first course, but to establish an early awareness that computation is not always sequential and that modern problems may involve multiple processors, distributed data, asynchronous events, and performance tradeoffs. Taken together, the implementations demonstrate that PDC can be introduced through incremental and locally appropriate changes rather than through a complete redesign of CS1.

1.2.1 Challenges

The most significant challenge was *finding instructional space in an already crowded CS1 curriculum*. Nearly every existing topic could be justified as useful, making decisions about what to condense difficult. The redesign therefore required instructors to distinguish between concepts that were essential for immediate student success and topics that were

specialized, repeated in later courses, or could be introduced with less detail. Even when no topic was eliminated completely, reducing practice or lecture time carried the risk that students would receive less reinforcement of traditional material. Careful alignment among course outcomes, assignments, laboratories, and later courses was necessary to ensure that the revised curriculum did not create gaps in students' preparation.

A second challenge involved *presenting PDC at an appropriate level for novice programmers*. Students in CS1 are simultaneously learning syntax, control structures, debugging, and problem-solving strategies. Introducing threads, race conditions, synchronization, or implementation-level details too early could create unnecessary cognitive load. Consequently, the revised course emphasized conceptual understanding, visual demonstrations, bounded code examples, and unplugged activities before asking students to interact with parallel code. The intent was not for students to become proficient parallel programmers in CS1, but for them to recognize that computational work can be divided, understand that modern computers contain multiple processing units, and begin reasoning about the potential benefits and limitations of parallel execution.

The redesign also required substantial *faculty effort*. Developing age-appropriate examples, identifying suitable locations in the existing curriculum, revising assignments, testing software and datasets, and repeatedly refining the materials required more than 100 hours across the initial development year. Coordination with laboratory instructors and teaching assistants added another layer of complexity because those supporting the course needed enough familiarity with the new concepts and technologies to guide students effectively. Technical differences among student computers, compiler configurations, OpenMP support, and execution timing also had to be anticipated. Although adopting already-developed modular materials can substantially reduce this workload, successful implementation still requires local adaptation, instructor preparation, and continued refinement based on student responses.

1.3 New Unplugged Activities

1.3.1 Flag Maker

1.3.1.1 Activity Summary

The Flag Maker unplugged activity was the most widely adopted activity created by our project. It has students take on the role of the processor to color cells of a grid in order to make the image of a flag. They do this sequentially and then in a variety of parallel configurations. The activity illustrates speedup and several concepts of coordination and contention in a graphical manner. It is a parallelized version of a graphical but non-parallel programming assignment to teach about loops by Emad [21]. The programming activity is described in §1.4.1.

This activity has been previously published [57]; the material on it here draws from that document.



Figure 1.1: Flag of Mauritius [65]

1.3.1.2 Prerequisites

The prerequisites for the Flag Maker activity are minimal. It has been used successfully as early as one third of the way into CS1 and should be suitable even for a CS0 class. The students need to understand the idea of operations executing one at a time and how loops allow repetition, but they do not even need conditionals or functions. The activity is built on the idea of the computer being able to perform multiple simultaneous operations, but we found that we could just tie this to the terms "dual-core" and "quad-core", with which the students were already familiar.

1.3.1.3 Learning Outcomes

Upon completing the activity, participants will be able to do the following:

1. recognize parallel computing terms such as speedup, contention, and pipelining;
2. recognize algorithmic solutions that are representative of data parallelism;
3. describe how parallel execution time is bounded by the slowest processor (i.e., the critical path).

1.3.1.4 Logistics

We split the class into groups of 5–6 or groups of 2–3, pairs of which merged to create groups of 5–6 after the first two phases of the activity. Each group needs 4 sheets of paper (1 per phase) with a large grid, each cell of which will be a single pixel in their drawing. (Information on the paper to use is in Section 1.3.1.7.) They also need one drawing implement of each of the flag's colors (red, blue, yellow, and green). The most popular implement is thick markers, but thin markers, bingo daubers, colored pencils, or crayons can also be used; expect unhappy students if they are provided with pencils or crayons, though giving these to some groups is a pedagogic choice discussed below. Each group will also need a way to keep time; we had students use the stopwatch program on their phones.

1.3.1.5 Running the Activity

Initially, the instructor introduces the activity by telling students they will take the role of computer processors to color a flag. Introduce the flag of Mauritius (see Figure 1.1), an island nation off the east coast of Africa. Split the class into appropriate groups and pass out the materials. It is also helpful to clarify the expectations on coloring: each person should color a single grid cell at a time (not scribble across a whole row) and to fill in the cells consistently. Consistency is the main requirement here, but we tend to encourage a style where each cell is colored pretty thoroughly, but not perfectly. (Perfection will greatly slow down the activity, but you want something that looks like the flag when completed.) It is helpful to show the class a sample flag that you've already colored.

Then, the activity proceeds in four scenarios, each with a different configuration of students coloring the flag. These scenarios are depicted in Figure 1.2, where P1–P4 mark the cells to be colored by up to four "processors" in the scenario. In the first scenario, one student (P1) in each group colors all the grid cells, one color at a time, while another student times them. After all the teams are done, have them report the times and record them on the board.

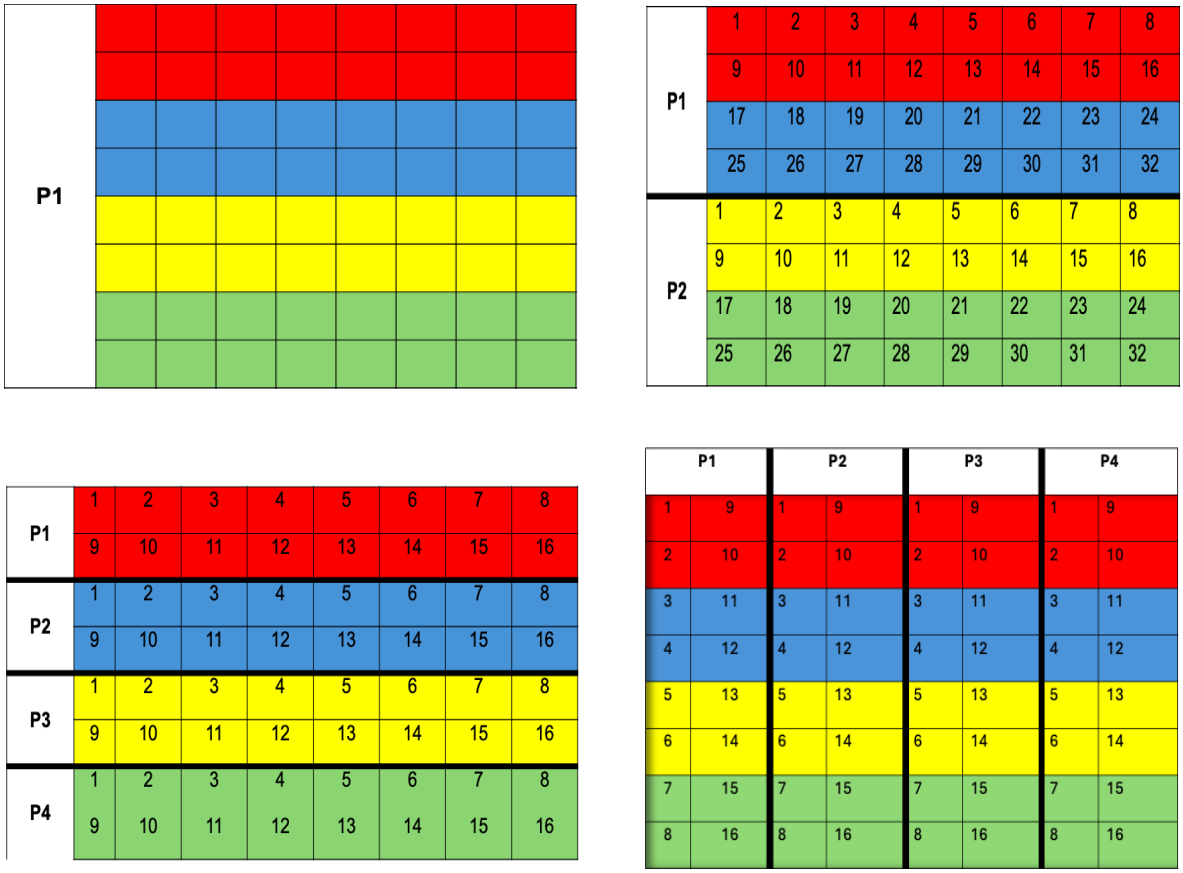


Figure 1.2: Graphical depiction of the Flag Maker scenarios: Scenario 1 (top left), Scenario 2 (top right), Scenario 3 (bottom left), and Scenario 4 (bottom right). P1–P4 depict the regions to be colored by each "processor". The cells after Scenario 1 are numbered to show the order in which they should be colored.

In Scenario 2 (top right), the flag is split in half, with one student (P1) coloring the red and blue stripes while another student (P2) colors the yellow and green stripes. In addition, there is a third student timing them; the time should be the finishing time of whichever student finishes last. After the coloring is completed, the times should again be recorded on the board.

Scenario 3 (bottom left) and Scenario 4 (bottom right) split the flag into 4 parts, each colored by one student, while a fifth student times them. Again, the time should be the finishing time of the last processor and the times should be collected on the board when all the groups have finished. In Scenario 3, each student colors one entire stripe. In Scenario 4, each student colors a vertical strip which includes two columns of each stripe. The cell numbering is also important here; each student is expected to color their first column, coloring red, blue, yellow, and green before returning to the top of their second column to repeat the colors. Note that each group should still have only one drawing implement of each color so the implements will have to be passed around.

1.3.1.6 Activity Discussion

Some institutions choose to discuss each scenario directly after each one, while others choose to have class discussion after the activity has ended. In both cases, the purpose of the discussion is to ask students what they observed and to put their observations into the context of parallel computing giving official terminology.

The first observation is likely that the coloring times decreased with the number of people doing the coloring. This leads to the definition of speedup (serial time divided by parallel time) and its calculation for each scenario. A particular term of interest is *linear speedup*, when the speedup is equal to the number of cores. More common is *sublinear speedup*, which occurs because the parallel program has to do extra work (like system calls to use all the cores) or some of them have to spend time waiting for a limited resource (like the markers in the activity. In parallel computing, extra work is *overhead* and resource-induced delays are *contention*. It is also possible to achieve *superlinear speedup*, one greater than the number of cores. In the activity, this may occur because students get more efficient with coloring as they practice or color less thoroughly as they get tired. Getting more efficient is somewhat like "warming up" a system, where a second run of the program can benefit from caching effects. In real systems, superlinear speedup is rare but can occur when performance is limited by memory rather than processing; running on multiple cores allows more cache space to be used, meaning that the problem may suddenly fit into the cache even though it didn't before.

1.3.1.7 Instructional Material

- TNTECH CS1 Flag maker presentation slides
- TNTECH CS1 Numbered grids to print for activity
- Blank grids to print for the activity. These are an alternative to the grids above; either can be used at instructor preference.
- A video demonstrating the potential speedup of coloring all pixels simultaneously instead of one at a time. This is framed as the difference between a CPU and GPU. We

suggest it as something the students could be asked to watch before the activity.

1.3.1.8 Example Implementations

The following institutions are using a variation of this assignment and you can find their details in their specific chapters:

- Knox CS1 - Section 3.3.1; Knox does post-activity discussion and also uses a Flag Maker plugged activity (discussed in Section 1.4.1).
- TNTECH CS1 - Section 2.3.2; TNTECH does discussion after each Scenario and has an OpenMP programming assignment 2.4.2 after the activity concludes.
- Webster CS1 - Section 6.2.2.1; Webster does a related flag-coloring activity/visualization.

1.3.1.9 Experience and Teaching Tips

We learned that students dislike using crayons for the Flag Maker activity and that markers are best. We also learned that this activity was best when there were at least 25 students in the class to participate and compete with each other. Friendly competition to see which team can finish first was found to naturally focus student's thinking on performance, and analyzing algorithmic approaches for ways to improve it. TNTECH lists their experience and teaching tips in more detail in §2.3.2.2.

1.3.1.10 Data and Analysis

TNTECH performed a pre-activity and post-activity quiz along with a post-activity survey. The results can be found in §2.3.2.3. Students showed some growth in learning some of the vocabulary associated with data parallelism. When asked what the most interesting thing they learned from the activity, almost a quarter of the students mentioned parallel processing, multicore processing, or simultaneous work. Other parallelism topics were also mentioned.

1.3.2 Unplugged Penny Search Activity

1.3.2.1 Activity Summary

The Unplugged Penny Sorting Search Exercise (PE) is an unplugged, hands-on active learning activity designed to introduce foundational concepts in parallel computing through a tangible, accessible task. Rather than relying on code or simulation, participants physically sort and search a collection of pennies to find the oldest one, enacting the roles of processors in a parallel system. The activity unfolds across three phases that progressively model a uniprocessor baseline, a balanced two-processor parallel execution, and a load-imbalanced five-processor scenario. By physically experiencing the difference in task completion time across these phases, participants develop an intuitive understanding of how parallel systems behave under varying conditions of data distribution and workload balance. Below is a more detailed description of the activity phases:

Phase 1: Uniprocessor phase At the data partitioning step, each group will receive a bag of pennies (e.g., 50), though the number can be reduced as long as the count remains an even number. In each group, Participant 1 performs sorting and searching for the oldest penny as their task on the allocated data (pennies), Participant 2 records time as their task and the other participants (if any) perform the task of observing. At a set time, all groups and group members begin their respective tasks.

- Participant 1 Task: find the oldest penny
- Participant 2 Task: tracks the time Participant 1 takes to complete the task and documents the results.
- Other participants: observe Participants 1 and 2. When they are done, shuffle coins and put them back in the bag.

Phase 2: Two Processor Parallel Phase At a set time, all groups will begin the tasks.

- Participants 1 and 2 of each group are given an equal number of pennies (e.g., 25). This workload or data distribution is managed by the instructor to create a load balanced parallel execution scenario.
- Participant 1 Task: Find the oldest penny in their bag.
- Participant 2 Task: Find the oldest penny in their bag.
- Participants 1 and 2 communicate and determine the overall oldest penny between both bags.
- Participant 3 Task: tracks the overall time for Participants 1 and 2 to determine the oldest penny in both bags.
- Any other participants: observe Participants 1, 2, and 3. When they are done, shuffle coins and put them back in the bag.

Parallel time is calculated as the time of the slowest participant among 1 and 2 and the communication time to determine the oldest penny of the two participants. When done observing, participants in the group shuffle coins and put them back in their respective bags. Each bag should have an equal number of pennies.

Phase 3: Five Processor Parallel Phase The activity facilitator or instructor gives each group four bags of two pennies for participants 1, 2, 3, and 4 and the remaining pennies (e.g., 42) to Participant 5. This uneven workload or data distribution introduces a load imbalance scenario for the parallel processes. At a set time, sorting begins for all groups.

- Participants 1-5 Task: Each must find the oldest penny in their bags. Once all participants complete their sorting, they must communicate with each other to identify the overall oldest penny.
- Participant 6 Task: tracks the overall time for Participants 1-5 to determine the oldest penny in their bags.
- Any other participants: observe Participants 1, 2, 3, 4, 5, and 6. When they are done, shuffle coins and put them back in the bag.

Parallel time is calculated as the time of the slowest participant among participants 1-5 and the communication time to determine the oldest penny of the five participants.

1.3.2.2 Prerequisites

This activity requires no formal prerequisites in parallel computing, making it suitable for introductory CS courses (CS1 and CS2) as well as upper-division systems courses where the instructor wishes to ground abstract concepts in concrete experience. Familiarity with basic algorithmic thinking, such as the idea that a search task has a measurable runtime, is sufficient preparation. The activity is also accessible to participants from non-CS disciplines participating in computing outreach or interdisciplinary courses.

1.3.2.3 Learning Outcomes

Upon completing the activity, participants will be able to:

1. explain the concept of data parallelism and how a dataset can be partitioned across multiple processors;
2. distinguish between load-balanced and load-imbalanced parallel workloads and articulate the performance implications of each;
3. describe how parallel execution time is bounded by the slowest processor (i.e., the critical path);
4. explain the concept of Amdahl's Law;
5. recognize the role of inter-processor communication in determining overall parallel performance; and
6. compare theoretical speedup with observed speedup from recorded timing data across the three phases.

1.3.2.4 Implementation Details

The PE activity is delivered as part of a self-contained instructional module on parallel computing. The module follows a deliberate pedagogical arc: it begins by motivating the relevance of PDC as a curriculum topic, transitions into conceptual grounding through short video resources, moves into the unplugged activity itself across three phases, and closes with structured group discussion and an optional survey. This arc is consistent whether the module is taught as a standalone guest lecture, a lab session, or an integrated class period within a broader CS1 or CS2 course.

The overall session outline is as follows: (1) motivation for why PDC belongs in the undergraduate curriculum, (2) an introduction to the concept of parallel computing through curated video resources, (3) the unplugged penny sorting activity across its sequential and parallel phases, (4) a group and class-wide discussion reflecting on observed results, and (5) an optional post-activity survey to capture student perceptions. Each component is described in detail below.

(i) Instructional Material (Lectures, Videos, Assignments, Tests, Discussions)

- **Motivation**: the session opens with a brief framing of why PDC is a required component of the undergraduate computing curriculum, grounded in the IEEE-CS TCPP Curriculum Initiative and the ACM/IEEE-CS 2013 Computer Science Curricula. Pointing students to the CDER's PDC curriculum rationale (available at `CDERcenter.o`

rg) situates the activity within a broader disciplinary context and helps students understand that parallel thinking is not an elective skill but a core competency expected of modern computing graduates. This framing takes no more than three to five minutes and can increase buy-in, particularly in courses where students may not have expected to encounter any PDC content.

- **Video resources**: before launching into the physical activity, a short video segment introduces the concept of parallel computing at an accessible level. Two videos are recommended for in-class use: a light, metaphor-driven introduction ("Digging Holes," available on YouTube) and a more substantive technical overview (available on YouTube). The LLNL parallel computing tutorial available (here) and Chris Bourke's webpage at University of Nebraska at Lincoln are recommended as post-class resources for students who wish to explore further. The two in-class videos together run approximately eight to twelve minutes and can be trimmed depending on available time. Their purpose is not to teach parallel computing comprehensively but to give students just enough conceptual vocabulary (processor, task, speedup, parallelism) to make sense of what they are about to enact physically.
- **Activity materials**: the primary instructional artifact is the activity itself, which requires minimal preparation overhead. One bag of pennies per group (approximately 50 pennies, though smaller or larger counts work if the count is even) is needed for Phases 1 and 3. For Phase 2, two bags per group containing an equal split (e.g., 25 pennies each) are required. Each group also needs a timer, and it is recommended that the instructor prepare an optional recording sheet on which groups log the elapsed time for each phase, the oldest penny identified, and any observations about coordination or idle time. This sheet doubles as a data collection instrument for the post-activity discussion. Note: if acquiring pennies is challenging, then instructors may also use custom index cards with random years printed on them, craft buttons with years printed on them using a permanent marker, or playing cards. April Crockett's TNTECH instructor presentation slides that walk through the scenarios and printable lab sheets are available on GitHub.
- **Assignments, tests, assessments**: To reinforce the concepts introduced through the activity, a short follow-up assignment or test can ask students to calculate the observed speedup from Phase 2 and Phase 3 using their recorded timing data, compare it against ideal (linear) speedup, and reason about the sources of deviation. Students can further be asked to connect their observations to Amdahl's Law by estimating the parallelizable fraction of the task and quantifying the overhead introduced by inter-processor communication. An optional post-activity survey (such as the ASPECT survey [70]) is recommended for capturing student perceptions of the activity's clarity, engagement, and perceived learning value. Survey data is particularly useful for instructors iterating on the activity across semesters or presenting the pedagogy at teaching focused venues.

(ii) How-To Guide

- Session structure: plan for at least a 50-minute class period. A suggested time allocation is: five minutes for PDC motivation, ten to twelve minutes for video viewing, twenty to twenty-five minutes for all three phases of the activity (including penny redistribution between phases), and ten to fifteen minutes for the group and class-wide

discussion. If time is limited, the motivation segment and one of the two videos can be reduced without compromising the activity itself.

- Before class: prepare one bag of approximately 50 pennies per group for Phases 1 and 3, ensuring each collection spans a wide range of years so that the search is non-trivial. Prepare two additional bags per group with equal penny counts for Phase 2 (e.g., 25 each). Print or otherwise distribute the optional recording sheet. Confirm that each group has access to a timer. If using the videos in class, test the links and audio in advance.
- Forming groups and running phases: divide the class into groups of five or six. Groups of four can participate in Phases 1 and 2 with minor adjustments but require at least five participants to enact Phase 3's load imbalance scenario meaningfully, since the contrast between participants with two pennies and the participant with the bulk of the coins is the core pedagogical mechanism of that phase. Run Phases 1, 2, and 3 according to the description provided above.
- Group and class discussion: after all phases conclude, facilitate a discussion at the group level first, then open it to the full class. Prompt students to share their recorded times, compare across groups, and articulate why Phase 3 did not improve on Phase 2 despite engaging more participants. Key observations to draw out include: the parallel runtime is bounded by the slowest participant; Participants 1 through 4 in Phase 3 were idle for most of the phase; and the act of comparing results across participants introduces communication overhead that is non-trivial even at small problem sizes. This is also a natural point to introduce or revisit Amdahl's Law formally.
- Assessments: conclude the session with a follow up assignment, test, or a post activity survey.

1.3.2.5 Experience and Teaching Tips

This activity runs effectively in a standard 50-minute or a 75-minute class period, with approximately 20 to 25 minutes devoted to all three phases and the remainder reserved for the post-activity discussion and assessments. In practice, Phase 1 tends to take the longest for individual participants and provides a useful baseline that makes the speedup in Phase 2 viscerally apparent. The dramatic slowdown (or absence of speedup) in Phase 3 due to load imbalance consistently generates spontaneous reactions from students, which makes the subsequent discussion highly engaging. One common facilitation challenge is that participants in the load-imbalanced phase (Participants 1-4) finish their two-penny assignments almost instantly and then sit idle while Participant 5 continues sorting. This idle time is itself a powerful teaching moment as students directly experience what it means for processors to be underutilized. Instructors should call attention to this idleness explicitly, as it connects naturally to the concept of processor starvation and the importance of effective scheduling. Timing precision varies across groups, and recorded times may differ more than expected from the theoretical speedup. Rather than treating this as a flaw, instructors can use the variance to discuss real-world sources of overhead: participants search at different speeds, communication takes time, and the act of comparing two pennies is not instantaneous. This mirrors the behavior of real parallel systems and reinforces why the theoretical speedup is an upper bound rather than a guarantee.

For larger class sizes, it is advisable to have a teaching assistant circulate across groups during Phase 2 and Phase 3 to ensure that the penny distribution is carried out correctly and that timing starts simultaneously across all groups. For very large lectures, the activity can alternatively be demonstrated at the front of the room with one volunteer group while the rest of the class observes and predicts the outcome of each phase before the timer is stopped. Finally, instructors should allow participants to observe the timing data first and then derive the intuition themselves before the formal definition of Amdahl's law is introduced, as this leads to considerably stronger engagement and retention. The activity works best as a discovery experience, not an illustration of a concept already explained.

1.3.2.6 Data and Analysis

This activity was originally proposed by Mary Smith and Srishti Srivastava, with consultation from Brett A Becker [59]. In the initial phase of the development of this activity, its implementation was primarily evaluated through a student engagement survey based on the Wiggins et al. (2017) ASPECT survey [70]. The modified ASPECT survey is presented in Figure 1.3.

This survey evaluates three key constructs: the perceived value of the activity, instructor contribution, and personal effort. The responses to each survey question were measured on a 5-point Likert scale, with values ranging from 1 (Strongly Disagree) to 5 (Strongly Agree). Faculty observational data were gathered during the students' participation in the activity.

The demographic variables, including gender, major, class standing, and age, were collected to explore possible relationships between these factors and engagement constructs. In summary, our analysis revealed that gender did not significantly impact the perceived value of the activity, instructor contribution, or personal effort across both examined activities. Age, however, showed a significant effect on instructor contribution in the PE activity. It is important to note that this initial analysis was conducted with a single group post-test and a small sample size, which may limit the generalizability of these findings. For a more in-depth understanding of this analysis, please refer to [59]. As the research evolved, the need to incorporate a pre- and post-survey approach is recognized to assess the extent to which the activities support student learning of key PDC concepts, such as concurrency, message passing, shared memory, scheduling, load balancing, speedup, and Amdahl's law.

Student Engagement Survey
(Based on Wiggins, Eddy et. al (2017) Assessing Student Perspective of Engagement in Class Tool (ASPECT))

Gender: Male ____ Female ____

Major: Computer Science _____ Engineering _____ Other: _____

Instructions: The following questions ask you about your experience with the **More Processors are Not Always the Best** activity that you completed today. Please rate how strongly you agree or disagree with each of the following statements. Circle your response

1) Explaining the material to my group improved my understanding of it.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

2) The instructor's enthusiasm made me more interested in the More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

3) Having the material explained to me by my group members improved my understanding of the material.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

4) Group discussion during the More Processors are Not Always the Best activity contributed to my understanding of the course material.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

5) The instructor put a good deal of effort into my learning for today's class.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

6) I had fun during today's More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

7) Overall, the other members of my group made valuable contributions during the More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

8) The instructor seemed prepared for the More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

9) I would prefer to take a class that includes this group activity over one that does not include this More Processors are Not Always the Best group activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

10) I am confident in my understanding of the material presented during today's More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

11) I made a valuable contribution to my group today.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

12) The instructor and TAs were available to answer questions during the More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

13) The More Processors are Not Always the Best activity increased my understanding of the course material.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

14) I was focused during today's More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

15) The More Processors are Not Always the Best activity stimulated my interest in the course material.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

16) I worked hard during today's More Processors are Not Always the Best activity.
1-strongly disagree 2-somewhat disagree 3-neither agree or disagree 4-somewhat agree 5-strongly agree

Figure 1.3: Modified ASPECT survey used for the Penny Sorting unplugged activity

1.3.3 Using Animations

1.3.3.1 Parallel Search Animation

Problem Description: Searching for a target value in a large collection is one of the simplest examples of an embarrassingly parallel problem. Similar to this idea: 1.3.2. This animation contrasts sequential and parallel search by allowing students to observe how dividing the search space among multiple workers can reduce execution time. See Figure 1.4. Students can adjust the number of workers and visualize how each worker is assigned a different portion of the dataset. The activity also introduces the idea that adding more workers does not always produce proportional speedup because of coordination overhead and workload imbalance.

The Animation can be found here: Parallel Search Animation

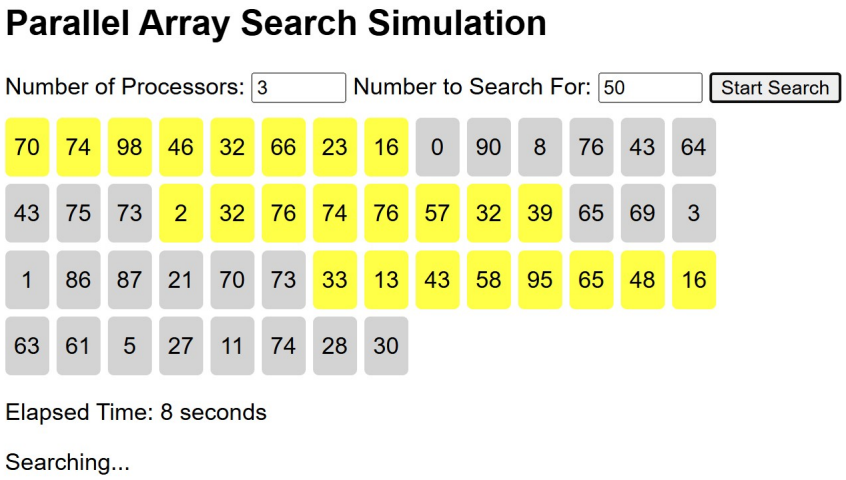


Figure 1.4: Parallel Search Animation

Prerequisites: Students should understand basic loops, arrays (or lists), and linear search. No prior knowledge of parallel programming is required.

Learning Outcomes: After completing this activity, students should be able to:

1. Distinguish between sequential and parallel search.
2. Explain how search tasks can be partitioned among multiple workers.
3. Describe why parallel search can reduce execution time.
4. Recognize factors that limit parallel speedup, including workload imbalance and synchronization overhead.

1.3.3.2 Parallel Linked List Animation

Problem Description Linked lists present a different set of challenges for parallel programming than arrays because elements are connected through pointers rather than con-

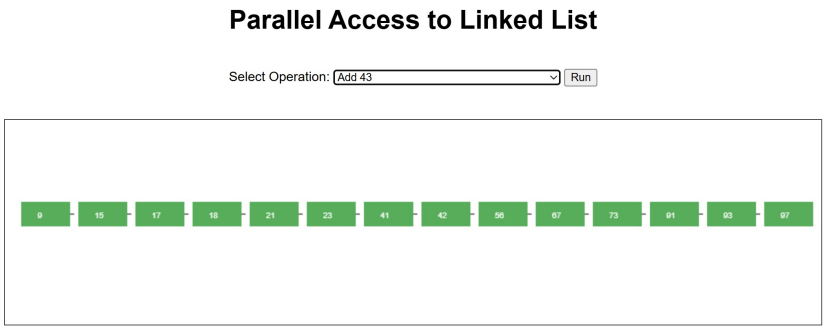


Figure 1.5: Parallel Linked List Animation

tiguous memory locations. This animation demonstrates how multiple operations, including searching, inserting, and deleting nodes, can occur concurrently on a shared linked list. See Figure 1.5. Students can execute multiple operations simultaneously and observe how concurrent modifications may interfere with one another when proper synchronization is not used. The animation introduces fundamental concepts such as concurrent data structures, synchronization, race conditions, and mutual exclusion intuitively and interactively.

The animation can be found here: Parallel Linked List Animation

Prerequisites: Students should understand basic linked list operations, including traversal, insertion, and deletion. A basic understanding of pointers or references is helpful, although no prior knowledge of parallel programming is required.

Learning Outcomes: After completing this activity, students should be able to:

1. Explain why concurrent operations on linked lists require synchronization.
2. Distinguish between read-only operations (search) and modifying operations (insert and delete).
3. Recognize how race conditions can arise when multiple threads access a shared linked list simultaneously.
4. Describe the role of mutual exclusion in maintaining data consistency during concurrent execution.

1.3.3.3 Flag Coloring Animation

Problem Description: The Flag Coloring animation introduces task decomposition through an interactive coloring activity inspired by the classic map-coloring problem. Similar to the unplugged activity described in Section 1.3.1, students are asked to assign different workers to color independent regions of a flag while ensuring that adjacent regions do not share the same color. See Figure 1.6. As students assign work to multiple workers, they observe that many regions can be colored concurrently, whereas neighboring regions introduce dependencies that require coordination. The activity provides an intuitive introduction to task decomposition, concurrency, synchronization, and dependency analysis without requiring any programming experience.

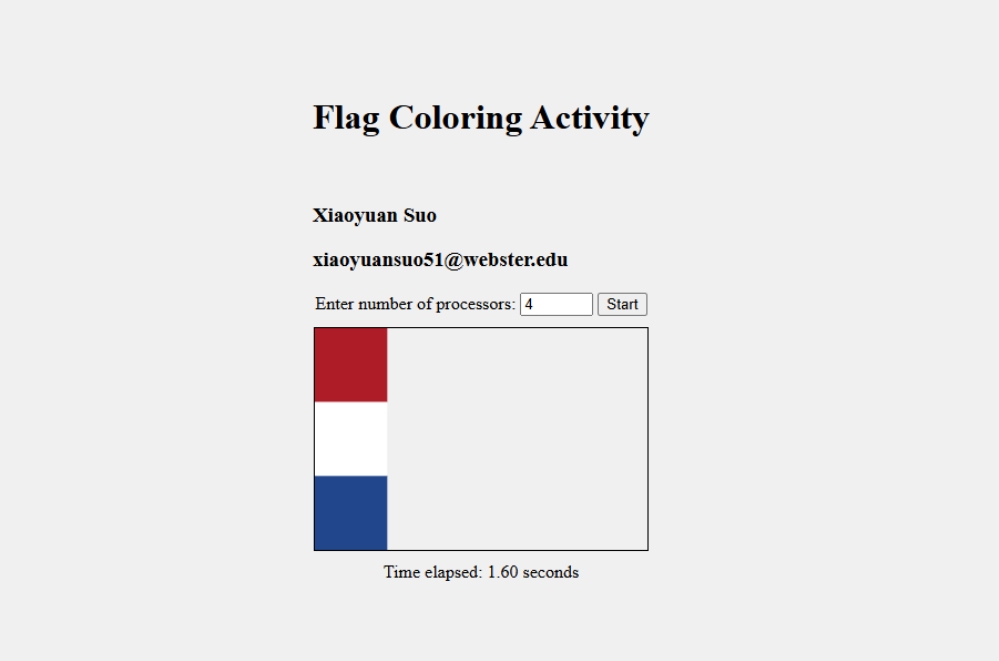


Figure 1.6: Flag Coloring Animation

The animation can be found here: flag coloring animation

Prerequisites: No programming experience is required. The activity is suitable for CS0, CS1, CS2, or non-computing audiences. Students only need a basic understanding of following rules and solving simple logical puzzles.

Learning Outcomes: After completing this activity, students should be able to:

1. Explain the concept of task decomposition by dividing a larger problem into smaller independent tasks.
2. Identify tasks that can be executed concurrently and tasks that require coordination because of dependencies.
3. Recognize that not all work can be performed in parallel due to shared constraints.
4. Relate graphical task decomposition to the design of parallel algorithms and workload distribution.

1.3.3.4 Zombie Attack Animation

Problem Description: The Zombie Attack animation introduces parallel computing concepts through a simulation of a zombie outbreak spreading across a two-dimensional grid. Similar to the programming activity described in Section 6.4.1, each cell in the grid represents an individual who may be healthy or infected. During each simulation step, infected individuals may spread the infection to their neighboring cells. See Figure 1.7. Students can divide the grid among multiple workers and observe how different regions are processed concurrently while neighboring partitions must exchange information at their boundaries. The

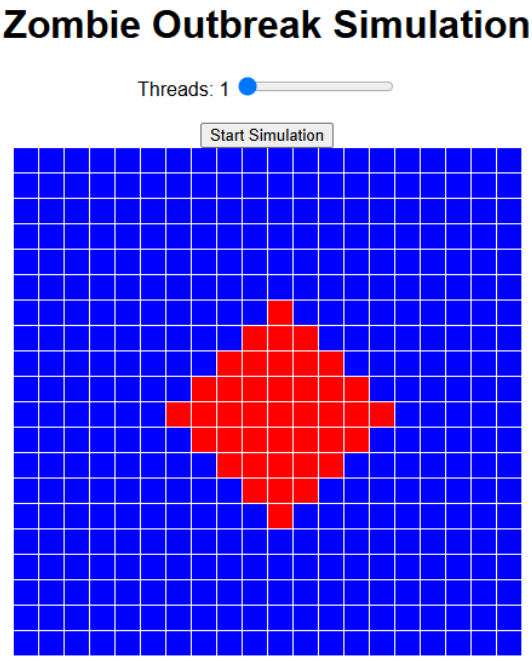


Figure 1.7: Zombie Attack Animation

activity provides an intuitive introduction to domain decomposition, parallel simulation, synchronization, and communication between neighboring tasks.

The animation can be found here: Zombie Attack animation

Prerequisites: Students should understand basic loops, conditional statements, and two-dimensional arrays (or matrices). No prior knowledge of parallel programming is required.

Learning Outcomes: After completing this activity, students should be able to:

1. Explain how a two-dimensional problem can be partitioned into smaller regions for parallel execution.
2. Describe the concept of domain decomposition and its role in parallel simulations.
3. Recognize that neighboring partitions must communicate to correctly update boundary cells.
4. Explain how synchronization ensures that all workers complete one simulation step before advancing to the next.
5. Relate the simulation to real-world parallel applications such as disease modeling, wildfire spread, weather forecasting, and image processing.

1.3.3.5 Experience and Teaching Tips for all the Animations

We found that animations were an effective way to introduce abstract PDC concepts before students encountered programming implementations. Compared with traditional unplugged activities, animations required substantially less classroom time while still promoting

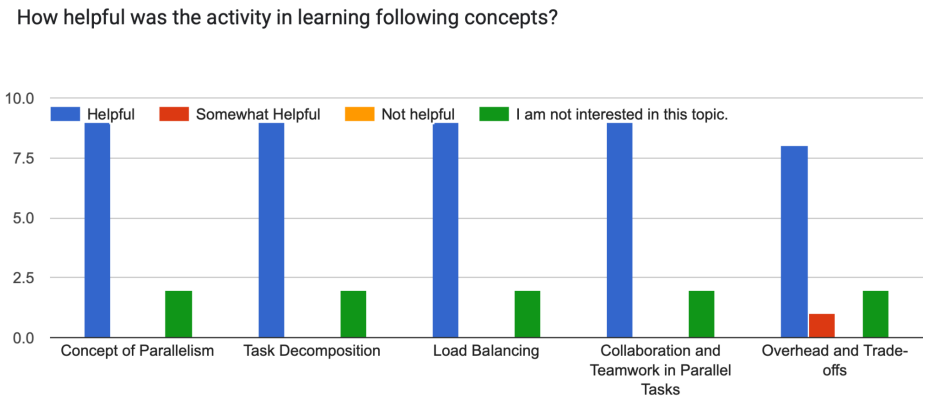


Figure 1.8: Participant Responses from Teaching Animations

discussion and conceptual understanding. Students generally found the visualizations engaging because they could immediately observe the effects of changing parameters, such as the number of workers, workload distribution, or decomposition strategy.

Animations worked particularly well when integrated with short instructor-led discussions rather than being presented as standalone demonstrations. Instructors were encouraged to pause the animation periodically and ask students to predict the outcome before continuing the simulation. These prediction activities often revealed common misconceptions about parallel execution and provided opportunities for immediate feedback.

Although the animations are designed to be self-contained, they are most effective when followed by a simple programming exercise or classroom discussion that connects the visualization to actual implementations. Instructors should also emphasize that animations intentionally simplify many aspects of parallel execution in order to highlight the underlying concepts rather than the implementation details.

More discussion can be found in a previous conference publication [62].

1.3.3.6 Data and Analysis

Survey data were analyzed using descriptive statistics to assess changes in students' perceptions and understanding of parallel computing concepts. Specifically, Likert-scale responses from the post-activity survey were summarized to gauge the perceived effectiveness and engagement of the unplugged activity, as shown in Figure 1.8.

Key findings from the post-activity survey include:

- 90% of participants agreed that the activity reinforced their learning and helped them understand how parallel computing is applied in real-world contexts.
- 81% of participants agreed that the activity increased their interest in parallel computing.
- 81% of participants indicated a preference for classes that incorporate group activities over those that do not.
- 90% of participants agreed that the activity was helpful in understanding key concepts

such as parallelism, task decomposition, load balancing, collaboration and teamwork in parallel tasks, and overhead and trade-offs.

Reflection responses were thematically coded to identify recurring themes related to conceptual understanding, engagement, and perceived value of the activity. The submissions to the Python parallel programming exercises were evaluated on the basis of precision, correct application of the parallel constructs, and clarity of implementation. Classroom observation notes provided additional qualitative evidence of student participation and collaborative interactions during the activities.

As shown in Figure 1.9, the students were then asked if they felt prepared to apply parallel computing concepts to new problems beyond this activity. 18.2 % strongly agreed with it. 54.5% agreed and 27.3 % were neutral.

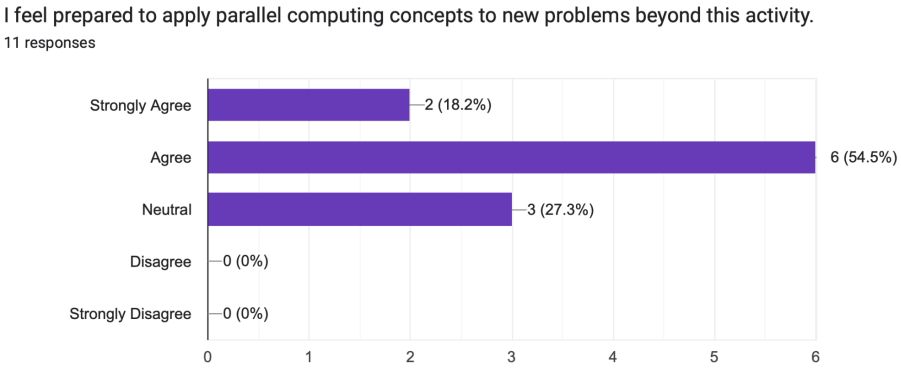


Figure 1.9: Participant Responses from Teaching Animations

1.3.3.7 Example Implementations

The following institution is using a variation of this assignment and you can find their details in their specific chapter:

- Webster CS1 - Section 6.3.1

1.4 New Plugged-in Activities

1.4.1 Programming Flag Maker

This is a version of the Unplugged Flag Maker activity (§1.3.1) that involves programming. It uses a new version of the given code, but the activity has long been used in CS1 at Knox College (and elsewhere; the idea came from Emad [21] at the University of Maryland College Park). It can be used with the unplugged activity in either order.

1.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

The Plugged-In Flag Maker Activity teaches nested loops by asking students to write code that draws flags on a 2D grid. For example, the student can write the following code to draw the Lithuanian flag as shown in Figure 1.10:

```
for (int r=0; r<grid.getNumRows()/3; r++)
{
    for (int c=0; c<grid.getNumCols(); c++)
    {
        grid.setColor(r, c, Color.YELLOW);
        grid.setColor(r + R/3, c, Color.GREEN);
        grid.setColor(r + 2 * R/3, c, Color.RED);
    }
}
```

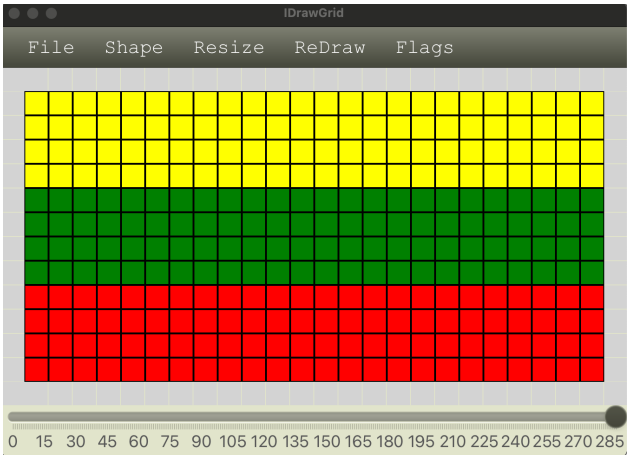


Figure 1.10: The Lithuanian flag, as rendered by the Flag Maker assignment.

The programming assignment is a series of empty method bodies for students to complete to draw flags of increasing complexity. Typically, the assignment begins with Poland, whose flag has 2 horizontal stripes, one of which is white and so it doesn't need to be explicitly drawn. (The background is white.) The code for this flag is included as an example. Next they draw a different flag with two stripes, then one with three stripes, then three stripes of unequal size, then one with vertical stripes, etc. Later flags have borders, crosses, triangles, and diagonal lines.

The visual aspect of this problem makes it easy for students to see if their code is working at a given size and a provided autograder runs the student code to create different-sized flags to ensure that it implements a general solution rather than one hard-coded to a particular size.

We provide a draggable "thumb" at the bottom of the user interface that allows students to replay the order in which grid squares are rendered. For example, Figure 1.11 shows the Lithuanian flag as rendered by the code in Figure 1.10. This code has one loop with 3 statements setting colors, which replays in a manner that looks parallel. If we implement

this same flag with 3 separate sets of nested for loops, the replay mechanism does not look like parallelism. Again, none of the code is parallel and we are not asking CS1 students to write code in parallel. The point is that this replay feature to demonstrate visually the concepts that underpin PDC.

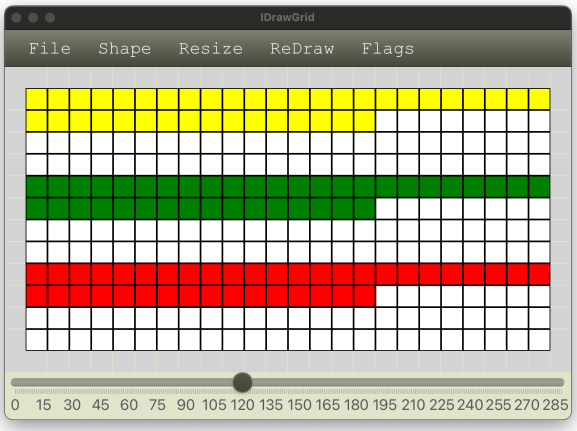


Figure 1.11: The draggable thumb at the bottom replays grid square placement. While not directly parallel, the thumb can nonetheless demonstrate parallelism.

1.4.1.2 Implementation Details

The Plugged-In Flag Maker Activity is based on an assignment from the mid-2000s originally created by Fawzi Emad [21], now a senior lecturer at the University of Maryland at College Park. The original assignment, written in Java, used Java Swing. The current version uses JavaFX, which has pros and cons.

In general, demand for desktop GUI applications written in languages like Java is shrinking because many desktop apps have been replaced by control panels that connect to cloud-based SaaS (Software as a Service) backends. For those applications, HTML, CSS, and JavaScript are the dominant medium for structuring and styling text-heavy dashboard applications with standard input/output widgets (radio buttons, text inputs, checkboxes, etc). There are now tech stacks that build desktop applications with raw HTML, CSS, and JavaScript, such as Electron, which bundles in 150–200 Mb of Chromium and Node.js to handle rendering, or Tauri, which uses the platform's native browser for rendering. Thus, while we use JavaFX for the implementation, the students do not see any of these details, and we keep as much of this hidden from students as possible.

Styling desktop apps in Swing is incredibly tedious because Swing has no equivalent to CSS and so styling must all be done programmatically; meanwhile, JavaFX can be styled with a flavor of CSS, but it differs from regular CSS in enough ways that it's unclear that it is worth learning outside of the specific use cases where a Java desktop GUI application is required.

- **Instructional Material (Lectures, Videos, Assignments, Tests)**

The Flag Maker assignment asks students to implement a variety of different flags using nested for loops. The assignment should provide students with a picture of the actual flag, and a picture of what we expect their code to generate. This is necessary, as we want students to simplify complex flags in the same way each time. For example, in Figure 1.12, we show the actual Japanese flag, as well as how we expect students to approximate the flag of Japan.
The Flag Maker code includes an "autograder" feature that runs each student's flag code for a variety of valid sizes for that flag, and compares these outputs to the correct output. We don't ship any code to produce the correct flag output programmatically with the starter code for the students; instead we include the solutions as text files that the autograder reads into an internal format using library code that is hidden from the students. This is not a perfect solution, but so far has worked well enough in practice. For example, we include a text file such as what is shown in Figure 1.13; The Flag Maker code reads this file into an internal format, then runs student code for the same flag size to produce the same internal format, and then the autograder diffs them and includes it in a report for faculty or TAs to read in order to speed up grading.

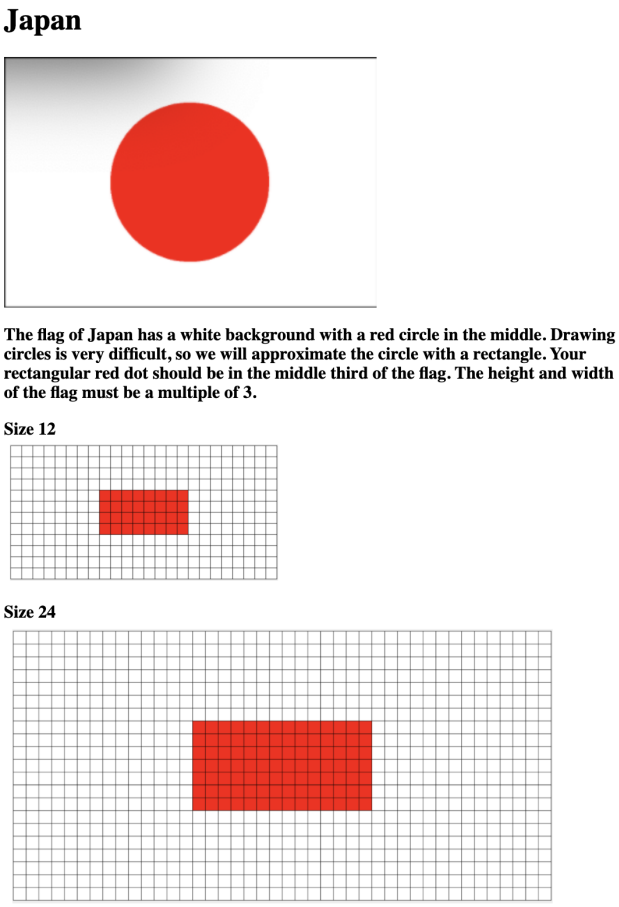


Figure 1.12: The assignment piece for the Japanese flag.

- **How-To Guides**

```
YYYYYYYYYYYY
YYYYYYYYYYYY
GGGGGGGGGGGG
GGGGGGGGGGGG
RRRRRRRRRRRR
RRRRRRRRRRRR
```

Figure 1.13: Solution as text for the Lithuanian flag.

Our main how-to guide is a YouTube video that explains how the Flag Maker system works.

1.4.1.3 Experience and Teaching Tips

The most important tip is to change the flags every term! If we don't change the flags, previous versions of the assignment will be submitted. Thus far, we have found that desperate students will submit a copy of an entire assignment from a previous term, but are less likely to get access to copies of a previous version of the assignment, read through it, and copy only the code for specific flags. Thus, some flags can be re-used from term to term, so long as the assignment is not identical to the previous several terms. We have a large collection of flags that we mix and match each term.

We provide infrastructure to automatically generate the assignment each term. Please contact the authors from Knox College with any questions about creating new flags.

The current implementation of Flag Maker uses JavaFX, which is no longer part of regular JVM releases. Instead, JavaFX is released through OpenJDK along with each JVM release. Since JavaFX is not part of the JVM, it requires different libraries on Windows, Mac Intel, Mac with Apple silicon, and Linux. For experienced developers, this is one setting in a pom.xml or build.gradle file; for novices we avoid using Maven or Gradle whenever possible and instead use the run button in VS Code.

For Flag Maker with JavaFX, we provide copies of the different libraries, and students copy the appropriate library into the `lib` folder in VS Code. Every year, some students don't read the directions or pay attention to instructions, and then copy the wrong library, or don't copy the library, or copy the library into the wrong place, and their code doesn't compile or run. If the assignment does not run correctly, faculty or TAs should first ask: Did you copy the libraries correctly?

Another challenge is that graders should look at the outputs from the autograder, rather than simply accepting the output. Students may choose a different but related color, such as red instead of orange, but otherwise generate the flag correctly; the autograder would flag this as a large number of different pixels, when in reality this is a simple misunderstanding. Similarly, students may be "off by one" in their flag, for example by having a triangular shape that is one column too wide or too narrow. That difference matters but represents one mistake, regardless of the number of differences between the correct version and the flag the student generated. To mitigate these challenges, we suggest using rubric-based grading for this assignment (i.e. 7/7 for totally correct, 5/7 for one conceptual mistake, 3/7 for two or more conceptual mistakes but code that is reasonable, 1/7 for some code written, etc)

rather than trying to measure the percentage of total grid squares that match exactly.

Can an LLM do this assignment? Yes, because LLMs can do nearly all tightly-specified CS1 assignments. However, if a student just gives the template code to the LLM and asks it to do the assignment, it will likely generate code that is suspiciously wrong. For example, as we see in Figure 1.12, we ask students to approximate the flag of Japan as a square. Unless students prompt the LLM otherwise, it will use methods like `Math.hypot` try to draw a circle, which is wrong for the assignment and represents knowledge beyond what a typical CS1 student would know.

1.4.1.4 Data and Analysis

Some data supporting the effectiveness of this assignment is given in §4.4.1.4.

1.4.2 Greenfoot Activity

1.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

Greenfoot [33, 67] is a visual IDE that empowers novices to build 2D games and simulations using Java code. It was developed by the team behind BlueJ [8] to facilitate an objects-first style of introductory Java instruction. A program in Greenfoot is called a *scenario*. Creating a game or simulation is actually quite simple and elegant: Students create agents (called *actors* in Greenfoot parlance) with an `act()` method that is called every frame to update the actor. Greenfoot provides helpful utility methods for detecting if two actors are touching or overlapping, changing background images, setting the position and rotation of an actor, changing the appearance of an actor, and so on. All Greenfoot classes are in a single package, so students simply `import greenfoot.*` to access all Greenfoot utility code.

Although Greenfoot is designed to be the IDE for a course, the idea behind our Greenfoot Activity is to introduce it as a game engine for a specific assignment late in the term. This allows us to introduce event-based programming and callbacks; as mentioned above, the code is attached to actors and called by the system rather than from a `main` method.

Figure 1.14 illustrates the basic Greenfoot IDE. Notice how the four actors (Phone, balloon, hedgehog, pizza) are clearly organized under the Actor, while the MyWorld world class (basically the screen where the scene takes place) is organized under World. Three of the four actors (hedgehog, balloon, pizza) can be seen in the actual scene; the Phone class does not show up in the scene intentionally to demonstrate for students that some code is required to create an instance of the Phone class and place it into the scene.

In Figure 1.15, we see the act() method to move an actor in Greenfoot. The code is simple and is accessible to CS1 students with less than a course of Java experience.

Given a few examples and their own creativity, novice students can create powerful games and simulations about two thirds of the way into CS1. Students have made basic 2D platformers, basic RPGs, space invader style shooters, puzzles, interactive chemistry study guides, and a wide variety of other projects. This module uses Greenfoot to teach PDC concepts and as a game engine for open-ended assignments, and has been previously published [24] as a Peachy Parallel Assignment [30].

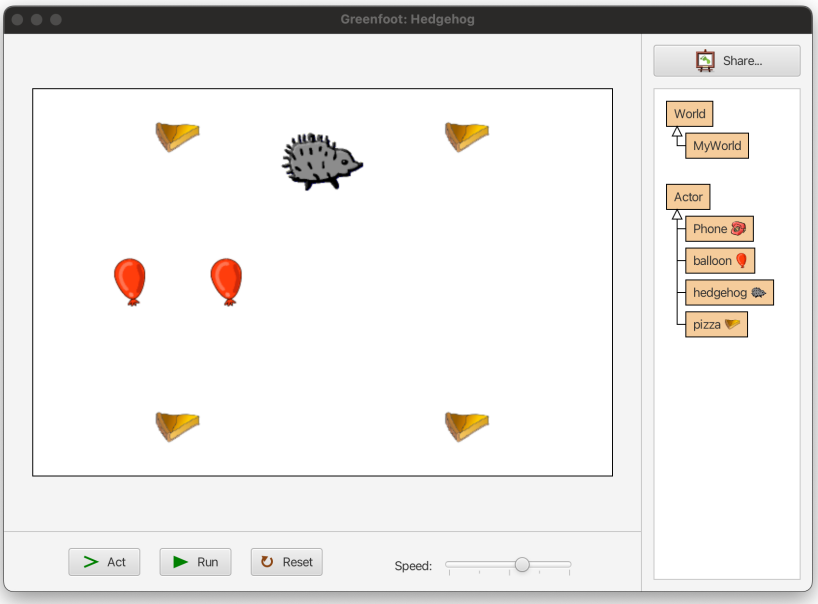


Figure 1.14: The Hedgehog example in Greenfoot. Notice how Actors are clearly organized in the IDE

1.4.2.2 Algorithms

The main PDC concept Greenfoot covers is event-driven programming. The `act()` method of each actor is a callback that the Greenfoot system calls each frame. By default, Greenfoot calls the `act()` methods of each actor in an unspecified order, which can lead to unpredictable behaviors. Figure 1.16 shows how to use the `setActOrder` method to define a consistent order in which the `act()` methods should be invoked. Most CS1 Greenfoot programs written by novices are unlikely to require this level of coordination to ensure correctness, but faculty can construct specific examples to demonstrate the challenge of ordering callbacks with side effects.

Another key concept Greenfoot teaches is that most modern programs do not start with a `main()` method typed into a blank screen. Instead, students must read examples and APIs to figure out how to plug their code into an existing framework that calls into their code when certain events happen. This helps teach students that, for many modern software development tasks, the *framework* is in charge, not the programmer. Game engines, even very simple ones like Greenfoot, are a fantastic way to highlight this reality for students in a relatively intuitive and highly engaging way.

1.4.2.3 Implementation Details

Instructional Material (Lectures, Videos, Assignments, Tests): We have assembled four complete Greenfoot examples and created instructional videos for each, in order to help faculty adopt Greenfoot in their CS1 course.

- Hedgehog: A simple game where the hedgehog needs to eat the pizza and avoid the

```
int speed = 3;
public void act(){
    if (Greenfoot.isKeyDown("right")) {
        setLocation(getX()+speed, getY() );
    }
    if (Greenfoot.isKeyDown("Left")){
        setLocation(getX()-speed, getY() );
    }
    if (Greenfoot.isKeyDown("Up")){
        setLocation(getX(), getY()-speed );
    }
    if (Greenfoot.isKeyDown("Down")){
        setLocation(getX(), getY()+speed );
    }
}
```

Figure 1.15: Example of how to move an actor in Greenfoot. Note the simplicity and elegance of the code.

```
public class MyWorld extends World {
    public MyWorld() {
        super(600, 400, 1);
        // Set execution order by Class
        setActOrder(Player.class, Enemy.class, Scoreboard.class);
    }
}
```

Figure 1.16: Example of setActOrder() method showing how to define a specific order for Greenfoot to call the act() methods.

balloons. Focuses on handling keyboard input, and detecting when two actors bump into each other (video).

- Shooting Game: Shoot birds that look suspiciously like butterflies with a cannon. Focuses on actors that spawn other actors (cannon spawns missiles), randomized movement of the birds/butterflies, actors change their sprites on collision (video).
- Cards: Very simple example that creates a deck of cards and deals them across the screen. Demonstrates how to have one actor (Card) that can have many sprites (the 52 playing card images) (video).
- Zelda: Core pieces of 1986 NES game The Legend of Zelda. Navigate a 2D grid and change backgrounds when leaving each screen, animated sprite, spawns enemies, very basic dialog window (video).

1.4.2.4 Experience and Teaching Tips

We have found two keys to successful Greenfoot assignments: First, trust the creativity of students, and second, embrace *open-ended, ill-formed* rather than *close-ended, well-formed* assignments [31].

Many introductory CS assignments are *well-formed*, *convergent*, and *close-ended* [20, 29]. CS1 assignments are tightly specified, ambiguities are eliminated whenever possible, and students tend to converge on one of a small set of correct solutions. For example, a typical convergent, well-formed assignment is to ask students to implement linked list methods.

Well-formed assignments have many strengths: they can be carefully designed to cover specific concepts, scoped to help students manage cognitive load, aligned with unit testing to enable autograding, integrated with worked examples, pruned of ambiguity to give students a clear definition of "done", and targeted so that specific misconceptions lead to predictable outcomes that are easy for instructors to diagnose. It would be hard to imagine introductory CS education without well-formed assignments.

However, well-formed assignments serve as cognitive scaffolding for novices that prepare students to solve "real world" problems, which are often messy, ambiguous, poorly specified, unpredictable, and frequently all of the above. Well-formed assignments provide minimal opportunities for students to flex their creativity, and apply their hard-won software engineering skills to open-ended problems.

Greenfoot naturally lends itself to open-ended, divergent, ill-defined assignments, as it would be uninteresting and impractical to ask all students to implement the same game in the same way. We typically ask students to make a game or simulation, and provide only minimal guidelines for how to do so. For example, we provide four complete worked examples, and ask students to use these as inspiration to make something new. We give guidelines about what students *must not* do (no minor tweaks or remixes of the examples, no tweaks of examples downloaded from greenfoot.org, etc), but minimal guidelines on what students *must* do. This is challenging for some students who expect very clear guidelines of what is required to receive a good grade, and so we have a couple of basic fallback options for those students, such as making a simple shooter or maze solving game. However, it has been our experience that many more students value the creative opportunities of Greenfoot far outweigh the challenges of students who seek the simplest pathway to an A on the assignment.

One concern with divergent, ill-formed, open-ended assignments is that some students do much more work than others, and that some students try to find a way to do the minimum. Our response: So what? Students often try to do the minimum, and part of teaching, advising, and mentoring is to encourage students to do more, not less. Different students do different amounts of work for a variety of reasons, such as more excitement or passion for a particular topic, work or class commitments outside of the class, whether a course is for a major or to fulfill a requirement or an elective, prior background in an area, confidence with course material, and so on. Well-formed, convergent assignments attempt to mandate a *minimum* threshold of work, but in the process they typically also set a *maximum* amount of work as well. Greenfoot assignments, at least in the way we implement them, take the opposite approach, and set minimal standards with no maximum, and then trust the students, and also our own ability to inspire, to challenge students to do more rather than

less.

One word of caution: The greenfoot.org website allows students to publish their scenarios, and then download and "remix" each other's work. This has resulted in an enormous library of a few classic assignments, many remixed with tiny modifications that make it difficult to find which one a panicked student downloaded and submitted right before the deadline. Because of this, we do not allow students to make versions of the *snake game*, *flappy bird*, or a couple of others that are well represented on greenfoot.org. There are just too many versions of these, and the search tools on greenfoot.org are not sophisticated enough to easily figure out which one a student submitted.

Greenfoot has been around since the late 2000s, and while it is commonly used in middle and high school, it is underutilized at the college and university level. The basic sprites included with Greenfoot are copyright free images (similar to Scratch [52]) that look appropriate for middle school, but may not present a sophisticated visual language appealing for a college audience. It is crucial not to judge the power and complexity of Greenfoot on these first impressions! It is quite easy to import custom images into a Greenfoot scenario, and we have never had complaints from college students about Greenfoot, or Scratch when we taught that years ago.

1.4.2.5 Data and Analysis

We did not collect data specifically on this assignment.

1.4.3 Plugged-In Earthquake Activity

1.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

The *Plugged-In Earthquake Activity* asks students to complete a C++ program that retrieves real, live earthquake data from the United States Geological Survey (USGS), filters that data, and displays selected earthquake information to the screen. Unlike a traditional introductory programming assignment focused only on loops, functions, and basic syntax, this assignment introduces students to the broader idea of working with remote, distributed, live data through an API. Students use a USGS earthquake data feed, returned in JSON format, to see how programs can interact with real-world data sources and support socially meaningful applications such as natural disaster monitoring.

The *prerequisites* for this assignment are that students should already have experience with several foundational C++ concepts, including for loops, arrays, functions, Makefiles, JSON, cURL, and distributed data. Students are not expected to build an API or JSON parser from scratch; instead, they are given starter code, supporting libraries, and Makefile templates so they can focus on understanding how data is retrieved, parsed, filtered, and displayed.

The main *learning outcomes* are that students will be able to develop and implement an algorithmic solution for interacting with a remote service, practice working with JSON-formatted data, and compile/run a program using a Makefile. By the end of the assignment, students should understand the client-server relationship at a basic level: their computer acts as the client, sends a request to a remote USGS server through an API, receives live

earthquake data, and then processes that data locally in a C++ program. A secondary learning outcome is conceptual. Students are asked to explain terms such as remote data, distributed data, API, and JSON, and to describe another real-world situation where large amounts of remote data are used to provide a public service. This documentation component reinforces the idea that computing is not only about writing isolated programs, but also about building systems that interact with live data sources in meaningful contexts.

1.4.3.2 Algorithms

The core algorithm for this assignment follows a retrieve–parse–filter–display structure. First, the program retrieves earthquake data from the USGS monthly earthquake feed. The data source is a live GeoJSON file containing detected earthquakes from the past month. Because the data is dynamic, the exact output may differ from one run to another depending on when the program is executed.

Second, the program parses the returned JSON data and accesses the features array. Each element in this array represents an earthquake event. For each earthquake, the relevant information is stored inside the properties object, including values such as magnitude, place, timestamp, and tsunami indicator.

1.4.3.3 Example Implementations

The following institutions are using a variation of this assignment and you can find their details in their specific chapters:

- TNTECH - CS1 course - Section 2.4.1
- Casper - CS1 course - Section 7.4.3
- UNL - CS1 course - Section 5.6.1

1.5 Course-wide Pre and Post Course Surveys

Development and testing teams administered the following common pre/post survey for CS1. The survey asked students their background knowledge in computing topics, programming languages, computing concepts, and PDC concepts. Students were asked to rate their prior experience on a 7-point Likert scale:

1. No Experience At All
2. Very Little Experience
3. Slightly Experienced
4. Between Slightly/Moderately Experienced
5. Moderately Experienced
6. Very Experienced
7. Extremely Experienced

The survey consisted of the following 26 items. Adopters can modify the pre/post surveys for background knowledge in computing topics, programming languages and computing concepts expected from the students in their courses. Similarly, depending on which PDC topics they

intend to introduce, they can choose parallel processing, distributed computing, and event handling.

1. Computer use experience (internet searches, email, discord, discussion groups, games, word processing, spreadsheets, organizing files & folders, etc.)
2. Programming Experience (self-initiated learning and/or formal programming course)
3. Programming Experience compared to your classmates
4. C
5. C++
6. C#
7. Java
8. JavaScript
9. PHP
10. Python
11. Scratch
12. Visual Basic
13. Other (any)
14. Data/Variable Types (integers, doubles, char etc.)
15. Arithmetic Expressions and Operators
16. Branching (if/else, relational and logical operators, relational expressions)
17. Loops (while, for, etc.)
18. Calling functions/methods
19. Arrays
20. Writing static functions/methods
21. Writing functions/methods
22. Writing classes
23. Creating objects
24. Parallel processing: Computing where the operations within a program are being executed simultaneously (i.e. running on different cores of the processor at the same time)
25. Distributed computing: Computing with a program that runs using data from a different computer (i.e. an online service)
26. Event handling: Having a function that responds to an independent action (i.e. a mouse click)

1.6 Conclusion

The activities in this chapter show that parallel and distributed computing can be introduced meaningfully in CS1. Across the institutional exemplars, the most successful approaches connect PDC ideas to familiar CS1 topics such as loops, arrays, searching, sorting, functions, objects, APIs, graphical interaction, and event handling. Unplugged activities, animations, visualizations, and short programming assignments help students first build intuition about workers, task decomposition, speedup, load imbalance, contention, synchronization, communication, remote data, and event-driven control before encountering more detailed implementation issues. A central lesson is that PDC is most effective in CS1 when it reframes and enriches existing course goals rather than appearing as disconnected additional content.

For instructors wishing to adopt, adapt, or extend these materials, the best starting point is often a small, easily integrated activity followed by a careful debrief that helps students name and interpret what they observed. Local constraints matter: class size, programming language, lab structure, student background, available teaching-assistant support, software setup, and instructional time all influence which activity should be chosen and how it should

be adapted. The institutional chapters, appendices, and repositories provide implementation details, slides, handouts, starter code, rubrics, surveys, timing guidance, and classroom evidence to support such adaptation. These instructional materials, assessment instruments, and evidence summaries are expected to be periodically curated and updated as additional adopters use, revise, and extend the activities in new CS1 contexts.

Chapter 2

CS1 Course Package – Tennessee Technological University

***Large-Section C++ Infusion with Unplugged Activities, OpenMP, and Remote Data*†**

April R. Crockett[1], Gerald C. Gannod[2]

[1]Tennessee Technological University, acrockett@tntech.edu
[2]Tennessee Technological University, jgannod@tntech.edu

†**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 68
2.1 Introduction . . . 72
2.2 How CS1 was changed and PDC topics integrated . . . 72
2.2.1 Integrated Syllabi (Pre and Post) . . . 73
2.2.2 PDC Topics Integrated . . . 73
2.2.2.1 Module 1 New PDC Content . . . 73
2.2.2.2 Module 2 New PDC Content . . . 74
2.2.2.3 Module 7 New PDC Content . . . 74
2.2.2.4 Module 8 New PDC Content . . . 74
2.2.2.5 Module 11 New PDC Content . . . 75
2.2.3 Highlights . . . 75
2.2.4 Challenges . . . 76
2.2.4.1 Preparing the Teaching Team for New Territory . . . 76
2.3 New Unplugged Activities . . . 76
2.3.1 Unplugged Penny Activity . . . 76
2.3.1.1 Instructional Material (Lectures, Assignments) . . . 78
2.3.1.2 Challenges . . . 79
2.3.1.3 Data and Analysis . . . 80
2.3.2 Unplugged Flag Maker Activity . . . 80
2.3.2.1 Instructional Material (Lectures, Assignments) . . . 81
2.3.2.2 Experience and Teaching Tips . . . 82
2.3.2.3 Data and Analysis . . . 82
2.4 New Plugged-in Activities . . . 82
2.4.1 Plugged-In Earthquake Activity . . . 82
2.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 82
2.4.1.2 Implementation Details . . . 84
2.4.1.3 Instructional Material (Lectures, Assignments) . . . 85
2.4.1.4 Data and Analysis . . . 86
2.4.1.5 Experience and Teaching Tips . . . 87
2.4.2 Plugged-In OpenMP Data Parallelism Activity . . . 88
2.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 88
2.4.2.2 Algorithms . . . 88
2.4.2.3 Implementation Details . . . 90
2.4.2.4 Instructional Material (Lectures, Assignments) . . . 90
2.4.2.5 How-To Guide . . . 90
2.4.2.6 Experience and Teaching Tips . . . 91
2.4.2.7 Data and Analysis . . . 91
2.5 Course-wide Pre and Post Course Surveys . . . 93
2.5.1 Results of Pre-Course and Post-Course Surveys . . . 94
2.5.1.1 Spring 2024 CS1 Baseline Results . . . 94
2.5.1.2 Fall 2024 through Spring 2026 CS1 Intervention Results . . . 95
2.5.1.3 Conclusion of Results . . . 95
2.6 Conclusion . . . 97
Appendix . . . 101

Chapter figures

2.1 Prior Programming Experience . . . 71
2.2 Organization of Pennies . . . 77
2.3 Penny Activity Phase 1 . . . 77
2.4 Penny Activity Phase 2 . . . 78
2.5 Penny Activity Phase 3 . . . 78
2.6 Penny Activity Phase 4 . . . 79
2.7 Box of 5,000 Pennies . . . 79
2.8 Level of Agreement for Unplugged Penny Activity . . . 80
2.9 Results of Flag Maker Pre-Quiz and Post-Quiz . . . 83
2.10 Results of Flag Maker Post-Activity Survey . . . 84
2.11 Screenshot of Earthquake Tracker Assignment . . . 85
2.12 Time to Complete Earthquake Assignment (All Semesters) . . . 86
2.13 Estimated Time to Complete Earthquake Tracker Assignment . . . 86
2.14 Level of Agreement Earthquake Tracker Assignment . . . 87
2.15 Screenshot of Plugged-In OpenMP Data Parallelism Assignment . . . 89
2.16 Hours Spent on Plugged-In OpenMP Lab Assignment . . . 92
2.17 What Students Learned from Plugged-In OpenMP Assignment . . . 93
2.18 Level of Agreement OpenMP Data Parallelism Lab Assignment . . . 93

Chapter tables

2.1 TNTECH CS1 Learning Outcomes & Blooms Levels . . . 69
2.2 Student Majors by Semester . . . 70
2.3 Student Gender Identity . . . 70
2.4 Race/Ethnicity of Students (Select All that Apply) . . . 70
2.5 Classification of CS1 Students . . . 70
2.6 Student Experience with General Computing . . . 71
2.7 Student Programming Experience Entering CS1 . . . 71
2.8 Spring 2026 Course Syllabus Calendar with Highlighted PDC Content . . . 73
2.9 Flag Maker Thematic Response to Student Interest in Activity . . . 83
2.10 Flag Maker Thematic Response to Student Feedback in Activity . . . 83
2.11 Sample Size for Pre-Course and Post-Course Surveys . . . 94
2.12 Baseline Results: Spring 2024 . . . 95
2.13 Intervention Results . . . 96

Abstract

This chapter presents a Computer Science introductory programming course (CS1) transformation at Tennessee Technological University (TNTECH) in the Department of Computer Science. This chapter also provides access to course materials so that other institutions may easily adopt and use the materials. Parallel and Distributed Computing (PDC) concepts were introduced into all sections of CS1 starting in Fall 2024 through lecture discussions, unplugged activities during lab class, and (plugged) programming assignments. The primary learning outcomes are for students to (1) recognize terminology associated with parallel and distributed computing topics, (2) identify algorithmic solutions that are representative of parallelism, (3) recognize when a problem solution requires interaction with a remote service, (4) develop programming solutions that use data parallelism, and (5) develop programming solutions that interacts with remote data.

Descriptor Elements

Programming Language Employed: C++

Relevant core courses: CS1

Pre-requisites: no prerequisites

Textbook: zyBooks: Programming in C++ by Frank Vahid and Roman Lysecky [66]

Relevant PDC topics and Blooms level: Below is a list of Parallel & Distributed Computing topics that we focused on when designing and creating materials for CS1.

- Data & Task Parallelism (Blooms levels: Understand, Analyze & Apply)
- Distributed Computing via Remote APIs (Blooms levels: Understand, Analyze & Apply)
- Event Handling (Blooms levels: Analyze)

Learning outcomes: The overall learning outcomes added to the CS1 course include the following:

- Recognize parallel computing terms such as data parallelism, task parallelism, task decomposition, speedup, scalability, and pipelining.
- Demonstrate the ability to identify algorithmic solutions that are representative of data parallelism.
- Implement algorithmic solutions that are representative of data parallelism.
- Recognize distributed computing terms such as remote data, distributed data, API, client / server, and JSON.
- Identify when a problem solution requires interaction with a remote service.
- Develop and implement algorithmic solutions for interacting with a remote service.
- Recognize when an algorithmic solution can benefit from the user of multiple threads that communicate via events or similar means of interaction.

Table 2.1 shows the activity-level learning outcomes added to Tennessee Tech's CS1 course with the incorporation of Parallel & Distributed Computing topics. Blooms levels are listed for each module and activity (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV), and Creating (CR)).

Table 2.1: TNTECH CS1 Learning Outcomes & Blooms Levels

PDC Concept	**Module Content & Blooms Level**							
	Mod 1 Slides 2.2.2.1	Unplug. Penny Activity 2.3.1	Mod 7 Slides 2.2.2.3	Earthquak Program 2.4.1	Mod 8 Slides 2.2.2.4	Unplug. Flag Activity 2.3.2	OpenMP Lab 2.4.2	Mod 11 Slides 2.2.2.5
PDC Vocabulary/Foundations	RE	UN	RE	UN	RE	UN		RE
Data Parallelism		RE			RE	UN	AP,EV	
Race Conditions/Synchronization	UN					UN,AN		
Speedup and Benchmarking							AP	
Distributed Computing			RE	AN,CR				
Event-Driven Programming								RE

Context for use: Tennessee Technological University (TNTECH) is a mid-sized public research university in Cookeville, Tennessee, United States. With an enrollment of about 10,000 students, TNTECH offers more than 40 undergraduate and 20 graduate degree programs across eight academic colleges and schools. The Department of Computer Science at TNTECH offers Bachelors, Masters, and Ph.D. degrees and consists of approximately 700 total students. At present, we have an average of over 200 students each semester enrolled in our CSC 1300: "Introduction to Problem Solving and Computer Programming", which is our CS1 course. The course is a 4 credit-hour lecture-lab, with 3 hours devoted to lecture and 1 hour devoted to lab. The course is usually split into two sections of lecture and eight to ten sections of lab; each lab section consists of approximately 25 students each. C++ is the programming language used for instruction, and this course is the first required computer science class in the curriculum of the Computer Science and AI undergraduate degree programs. The CS1 course is taken by students in several additional majors as well, as shown in Table 2.2. Eighty percent of the students in CS1 identify as male and approximately 15% identify as female as shown in Table 2.3 and student race is shown in Table 2.4. Table 2.5 shows the classification of students taking the CS1 course. Students enter CS1 having a wide variety of prior experience in computing and programming. Table 2.6 shows incoming student experience of general computing such as internet searches, email, discord, discussion groups, games, word processing, spreadsheets, organizing files, and organizing folders. Table 2.7 shows incoming programming experience of students entering CS1. This diversity includes exposure to different programming languages, variations in depth, and no programming experience whatsoever. We use the student-selected Experience-Based Grouping (EBG) model to address experience diversity and improve academic success for all students [14–17]. The variation of experience and backgrounds extends to having students from several class standing levels and majors in the course.

Table 2.2: Student Majors by Semester

Major	Spring 2024	Fall 2024	Spring 2025	Fall 2025	Spring 2026	Total
Computer Science	53	153	74	109	32	**421**
Electrical Engineering	36	36	39	62	42	**215**
Mechanical Engineering	19	12	29	21	37	**118**
Computer Engineering	10	10	18	24	18	**80**
Mathematics	4	4	6	10	3	**27**
Physics	1	6	2	10	6	**25**
Music (Live Audio Engineering)	2	2	1	5	1	**11**
Business Information Technology	1	2	0	2	0	5
Civil Engineering	0	1	1	3	0	5
Computer Science Education	1	0	1	1	1	4
Basic Engineering	1	0	0	1	0	2
Business	0	0	1	0	1	2
Interdisciplinary Studies	0	0	0	2	0	2
Manufacturing Eng. Tech.	1	0	0	0	1	2
Nuclear Engineering	0	0	1	1	0	2
Chemical Engineering	0	0	0	0	1	1
Other	0	1	1	6	0	10
Total	**129**	**227**	**174**	**257**	**145**	**932**

Table 2.3: Student Gender Identity

Gender	Spr 24 n=159	Fall 24 n=226	Spr 25 n=168	Fall 25 n=235	Spr 26 n=171	Total n=959
Male	124	181	128	192	139	764 (79.7%)
Female	26	33	31	33	22	145 (15.1%)
Other	3	6	3	5	2	19 (2.0%)
No Response	6	6	6	5	8	31 (3.2%)

Table 2.4: Race/Ethnicity of Students (Select All that Apply)

Race/Ethnicity	Spr 24 n=159	Fall 24 n=226	Spr 25 n=168	Fall 25 n=235	Spr 26 n=171	Total n=959
White	120	167	130	171	121	709 (73.9%)
Black/ African American	19	23	17	17	16	92 (9.6%)
Hispanic	7	15	15	24	15	76 (7.9%)
Asian/ Pacific Islander	13	30	8	16	13	80 (8.3%)
American Indian/ Alaska Native	0	5	2	1	1	9 (0.9%)
Middle Eastern/ North African	0	4	2	0	1	7 (0.7%)

Table 2.5: Classification of CS1 Students

Classification	Spr 24 n=159	Fall 24 n=226	Spr 25 n=168	Fall 25 n=235	Spr 26 n=171	Total n=959
First year (freshman)	100	119	102	125	120	566 (59.0%)
Sophomore	43	78	52	74	39	286 (29.8%)
Junior	13	25	11	30	8	87 (9.1%)
Senior	3	3	3	6	4	19 (2.0%)
Graduate student	0	1	0	0	0	1(0.1%)

Table 2.6: Student Experience with General Computing

General Computing Experience	**Spr 24** n=159	**Fall 24** n=226	**Spr 2025** n=168	**Fall 25** n=235	**Spr 26** n=171	**Total** n=959
Extremely	42	52	37	54	39	224 (23.4%)
Very	63	82	61	76	64	346 (36.1%)
Moderately	44	56	46	58	47	251 (26.2%)
Between Slightly and Moderately	0	16	11	26	10	63 (6.6%)
Slightly	8	10	9	16	5	48 (5.0%)
Very Little	0	8	2	4	4	18 (1.9%)
None At All	2	2	2	1	2	9 (0.9%)

Table 2.7: Student Programming Experience Entering CS1

Prior Programming Experience	**Spr 24** n=159	**Fall 24** n=226	**Spr 25** n=168	**Fall 25** n=235	**Spr 26** n=171	**Total** n=959
Extremely	2	3	1	8	0	14 (1.5%)
Very	9	18	12	18	9	66 (6.9%)
Moderately	50	48	37	46	34	215 (22.4%)
Between Slightly and Moderately	0	30	32	26	40	128 (13.3%)
Slightly	68	40	36	43	29	216 (22.5%)
Very Little	0	59	35	61	47	202 (21.1%)
None At All	30	28	15	33	12	118 (12.3%)

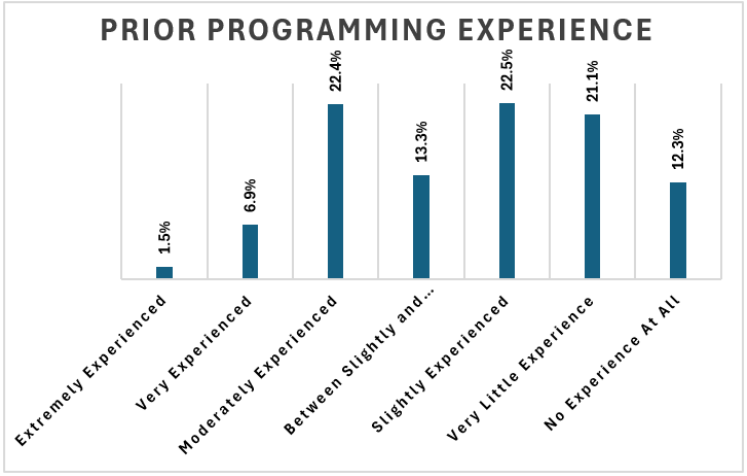


Figure 2.1: Prior Programming Experience

2.1 Introduction

Modern computing increasingly relies on parallelism, distributed systems, and data-intensive workflows. National curriculum recommendations such as the ACM/IEEE Computer Science Curricula and the IEEE TCPP Parallel and Distributed Computing (PDC) guidelines, have highlighted the growing relevance of these topics within undergraduate computing education [35, 51]. At Tennessee Technological University (TNTECH), we began infusing PDC concepts into our undergraduate computing curricula starting in Fall 2024 with our CS1 course.

§2.2 below describes in detail what was changed in our CS1 course and what specifically was added. We started spending less time on some topics in order to make room for parallelism, distributed computing, and event handling. We also modified some assignments and activities to incorporate PDC topics. §2.2.3 highlights the changes and §2.2.4 discusses challenges faced in updating the curriculum. The details on all new activities are in sections 2.3 and 2.4.

2.2 How CS1 was changed and PDC topics integrated

The CS1 course was revised to introduce parallel and distributed computing (PDC) concepts without fundamentally changing its existing purpose or removing the programming foundations that students need for later courses. Rather than treating PDC as a separate, advanced unit, the course incorporated these ideas into familiar CS1 topics, including computer hardware, loops, arrays, vectors, file processing, and problem decomposition. Approximately four hours of lecture and laboratory time were devoted specifically to PDC content, distributed across the semester so that students encountered the concepts in contexts where they naturally complemented the existing curriculum. This approach allowed students to begin developing a modern model of computation while they were still learning conventional sequential programming.

Some existing topics were condensed to create space in the course. Less instructional time was devoted to detailed coverage of primitive data types, two-dimensional arrays, C-style strings, binary and random-access files, and extensive output formatting. These topics were not necessarily removed entirely; instead, they were treated more selectively because they were considered specialized, less central to contemporary programming practice, or appropriate for reinforcement in later courses. Foundational topics such as variables, selection structures, repetition, functions, arrays, vectors, classes, and file input remained central to the course. The goal was therefore not to replace established CS1 content with PDC, but to periodically reevaluate the relative emphasis placed on different topics and make room for concepts that better reflect modern computing environments. Specifically, a programming assignment formerly centered on arrays and vectors was revised so that students accessed live remote earthquake data, while a lab that had previously focused only on arrays was updated to include OpenMP, and other lab classes incorporated unplugged activities on parallel programming concepts that are described in §2.3.

2.2.1 Integrated Syllabi (Pre and Post)

The course calendar from the syllabus for CS1 at TNTECH as shown in Table 2.8 spans 16 weeks across eleven content modules, covering the standard CS1 arc from introductory programming and C++ fundamentals through branching, loops, functions, arrays, pointers, C++ structs, and classes. The 16th week is final exams week. The PDC interventions, highlighted in the course calendar, are threaded into this existing structure at three points, each chosen because the underlying CS1 content provides a natural and mutually reinforcing context for the PDC concept being introduced.

Table 2.8: Spring 2026 Course Syllabus Calendar with Highlighted PDC Content

Module	Week	Assignments	PDC Content Added	Quiz
Mod 1: Introduction	Weeks 1 & 2	Syllabus, Informed Consent Form, Pre-Course Experience Survey, zyBook Chapter 1, Group Selection Form	Informed Consent Form, Pre-Course Experience Survey, zyBook Section over Parallel Programming, Slides (Mod 1 PDC Content in §2.2.2.1)	Mod 1 & 2 Quiz
Mod 2: Prog. Basics	Weeks 2 & 3	zyBooks Chapter 2, Get Computer Set Up, Lab 1 Assignment	Unplugged Penny Activity (Mod 2 PDC Content in §2.2.2.2)	Mod 1 & 2 Quiz
Mod 3: Data Types & Math Expressions	Weeks 3 & 4	zyBooks Chapter 3, Lab 2 Assignment, Program 1 Assignment	N/A	Mod 3 Quiz
Mod 4: Branching	Week 5	zyBook Chapter 4, Lab 3 Assignment	N/A	Mod 4 Quiz
Mod 5: Loops	Week 6	zyBook Chapter 5, Program 2 Assignment, Lab 4 Assignment	N/A	Mod 5 Quiz
Mod 6: Files	Week 7	zyBook Chapter 6, Lab 5 Assignment	N/A	Mod 6 Quiz
Mod 7: Functions, Distributed Computing	Week 7 & 8	zyBook Chapter 7, Lab 6 Assignment, Lab 7 Assignment, Program 3 Assignment	Slides, Plugged-In Earthquake Activity(Mod 7 PDC Content in §2.2.2.3)	Mod 7 Quiz
Mod 8: Arrays, & Parallel Prog.	Week 9 & 10	zyBook Chapter 8, Lab 8 Assignment, Lab 9-OpenMP Data Parallelism Lab Assignment, Program 4 Assignment	Slides, Plugged-In OpenMP Data Parallelism Assignment, Unplugged Flag Maker Activity (Mod 8 PDC Content 2.2.2.4)	Mod 8 Quiz
Mod 9: Pointers	Week 11 & 12	zyBook Chapter 9, Lab 10 Assignment, Program 5 Assignment	N/A	Mod 9 Quiz
Mod 10: ADTs & Structures	Week 13 & 14	zyBook Chapter 10, Lab 11 Assignment	N/A	Mod 10 Quiz
Mod 11: Classes & Objects, Event Handling / GUIs	Week 14 & 15	zyBook Chapter 11	Slides (Mod 11 PDC Content in §2.2.2.5) & GUIs	N/A

2.2.2 PDC Topics Integrated

2.2.2.1 Module 1 New PDC Content

In our CS1 class, computer hardware, the power wall, multiple cores, and introducing parallel programming was discussed during week 1 of lecture. We also assigned the informed

consent form and the pre-course experience survey during week 1 of the course. We use a zyBook section created by Casper College (§7.3.1) over Parallel Computing and ask a few questions as part of the zyBook chapter homework. Here is a list of PDC materials that were used in Module 1:

- PRESENTATION SLIDES_Module 1 Introduction.pptx
- CS1 & CS2 Spring 2026 Informed Consent Form
- CS1 Spring 2026 Pre-Course Experience Survey
- zyBook Chapter 1.5 Parallel Computing

2.2.2.2 Module 2 New PDC Content

Students go to lab class for 1 hour and 15 minutes once a week starting in week 2. Each lab section is taught by undergraduate teaching assistants. During their first lab, the TAs cover lab policies and then have the students participate in the **Unplugged Penny Activity**, discussed further in §2.3, which helps students get to know each other since it is a group activity and further introduces students to parallel programming concepts.

2.2.2.3 Module 7 New PDC Content

After covering functions in lecture, we introduce students to distributed computing concepts including:

- getting data from a remote server,
- JSON format and Niels Lohmann's JSON for Modern C++ library [39],
- cURL library [61],
- Make utility and Makefiles [36]

We typically cover the topics above in one 55-minute lecture. Then, students are assigned the **Earthquake Tracker Programming Assignment**. Here is a list of PDC materials that were used in Module 7:

- PRESENTATION SLIDES_Module 7 Functions & Distributed Computing.pdf
- PRESENTATION SLIDES_Makefiles_CSC 1300.pptx
- Plugged-In Earthquake Activity(§2.4.1)

2.2.2.4 Module 8 New PDC Content

Module 8 begins with arrays and the STL Vector class. Then, we introduce parallel programming and discuss the differences between task and data parallelism. This leads into a lecture introducing students to the OpenMP API [3, 19] for parallel programming in C++. During the following lab class, students participate in an **Unplugged Flag Maker Activity** and then complete the **Plugged-In OpenMP Data Parallelism Assignment** for students to practice what they have learned. Here is a list of PDC materials that were used in Module 8:

- PRESENTATION SLIDES_Module 8 Purple_Arrays Vectors Data Parallelism.pptx
- PRESENTATION SLIDES_OpenMP Introduction.pptx

- Parallel Programming & Chrono Program Examples
- Unplugged Flag Maker Activity(§2.3.2)
- Plugged-In OpenMP Data Parallelism Assignment (§2.4.2)

2.2.2.5 Module 11 New PDC Content

Module 11 introduces students to Classes & Objects as well as Event Handling & Graphical User Interfaces (GUIs). This module covers ideas and terminology such as event, response, synchronous vs. asynchronous event handling, and GUIs. During lecture, the instructor also demonstrates UNL's Asynchronicity webpage. The Event Handling part of lecture usually takes about 30 minutes to cover. Here is a list of PDC material that was used in Module 11:

- PRESENTATION SLIDES_Module 11_OOP & Event Handling.pptx
- UNL Asynchronicity Demonstration

2.2.3 Highlights

Much of the existing CS1 course remained unchanged following the integration of PDC content. The course continued to emphasize the foundational programming concepts needed for success in subsequent courses, including variables, selection structures, loops, functions, arrays, vectors, pointers, structures, classes, and file input. The overall sequence of the course modules also remained largely the same. Rather than adding a separate PDC unit, the new material was distributed across the semester and connected to topics already being taught.

The most significant changes involved adjusting the amount of time devoted to selected existing topics and modifying several instructional activities. Coverage of primitive data types, extensive output formatting, two-dimensional arrays, C-style strings, and specialized file-processing techniques was condensed to create approximately four hours of lecture and laboratory time for PDC-related instruction. A traditional arrays-and-vectors programming assignment was replaced with the Earthquake Tracker Programming Assignment, which introduced remote data retrieval and distributed computing concepts. An existing arrays laboratory was expanded into the OpenMP Data Parallelism Lab, and unplugged activities were added to illustrate parallel search, workload division, and the effects of using multiple processors.

A key strength of the redesign was that each PDC concept was introduced where it naturally complemented the existing curriculum. Parallel computing was initially connected to computer hardware and multicore processors, distributed computing was introduced alongside functions and file processing, data parallelism and OpenMP were paired with arrays and vectors, and event handling was presented with classes, objects, and graphical user interfaces. This organization allowed students to encounter modern computing concepts without sacrificing the course's primary emphasis on introductory programming fundamentals. It also made the interventions modular, allowing instructors to adopt individual activities or assignments without completely restructuring the course.

2.2.4 Challenges

2.2.4.1 Preparing the Teaching Team for New Territory

A challenge of teaching the new material was that the material was not only new to the students in the course, but also to the team of teaching assistants. This meant that we needed to find time to teach the concepts, activities, and lab assignments to the teaching assistants before teaching the students. Finding the time to do this was definitely challenging. There were also some teaching assistants that didn't understand why these concepts were being taught in CS1 and thought it was too much to ask CS1 students to learn about and practice these concepts. We originally didn't expect this confusion and didn't explain the reasoning. However, in the 2nd semester of implementing the PDC concepts we knew that explaining the reasoning must be part of the TA training.

2.3 New Unplugged Activities

During the first week of lecture, students are introduced to programming concepts and computer hardware. As part of the discussion of the central processing unit, we examine the historical Von Neumann architecture, which relies on a single CPU, and the efforts made to improve computational efficiency and speed by increasing the number of transistors. We then discuss how these improvements eventually encountered the power wall, motivating the shift toward multicore architectures. Finally, students learn how multicore processors enable parallel computing, in contrast to traditional serial computing. To solidify this idea, we introduced the **Unplugged Penny Activity** in lab.

2.3.1 Unplugged Penny Activity

Our first lab class in week 2 (out of 16 weeks) begins with the Unplugged Penny Activity, which is introduced in §1.3.2. Figure 2.2 shows how we store the pennies and distribute during the activity. This is a great unplugged activity for the first lab because it allows students to get to know each other and it reinforces the concept of parallelism that has already been introduced in lecture (shown in §2.2.2.1.

At TNTECH, we show presentation slides during each phase of the activity and we added an additional phase that is not described in §1.3.2.

Phase 1 is shown in Figure 2.3. After all groups of students complete Phase 1, the instructors describe that phase 1 represents a purely sequential algorithm where there is only one thread (or worker) performing the task. They explain that there is no parallelism, and the task is completed by single worker from start to finish. The term "single-core processing" is introduced. We then ask students, "If we were to divide the pennies up in the beginning and have multiple sorters, do you think this would take more or less time?" and then we proceed to Phase 2.

Phase 2 is shown in Figure 2.4. After all groups of students complete Phase 2, the instructors describe that phase 2 represents two independent tasks (sorting two separate bags) are executed concurrently. We further explain that this represents task parallelism where multiple workers perform tasks in parallel and then synchronize (compare results) at



Figure 2.2: Organization of Pennies

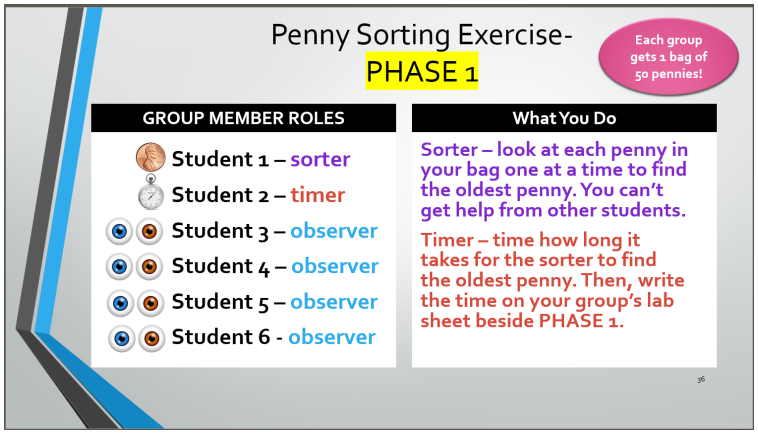


Figure 2.3: Penny Activity Phase 1

the end. The synchronization step is the comparison of the oldest pennies from the two bags, which ensures that the final result is the overall oldest penny. Students are then informally asked the following questions:

1. How did dividing the task between two people (parallel sorting) affect the overall time compared to Phase 1 (sequential sorting)? Did you experience speedup?
2. If each person worked at different speeds, how might that affect the overall time it takes to find the oldest penny?
3. Would adding more people (more sorters with smaller bags of pennies) make this task faster? Why or why not?

The second question typically leads to a discussion about load balancing and how the performance of parallel tasks can be uneven if processors work at different speeds. The third question can encourage students to think about scalability in parallel programs and the potential diminishing returns when adding more workers.

Phase 3 is shown in Figure 2.5. After all groups have completed Phase 3, the instructors talk about how this phase demonstrates data parallelism, where the dataset (pennies) is

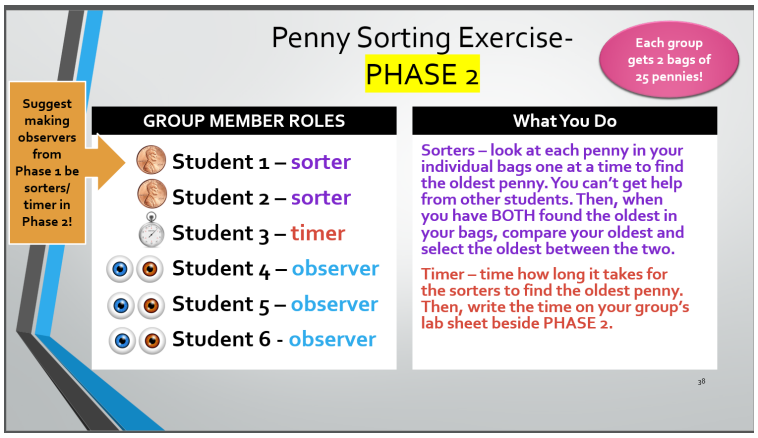


Figure 2.4: Penny Activity Phase 2

divided among multiple workers. We point out, however, that there is load imbalance, as one student has significantly more pennies to sort than the others. This imbalance reflects real-world scenarios where tasks are unevenly distributed among parallel workers, which can lead to inefficiencies. After finding the oldest penny, the students must communicate (synchronize) to combine their results and identify the oldest penny.

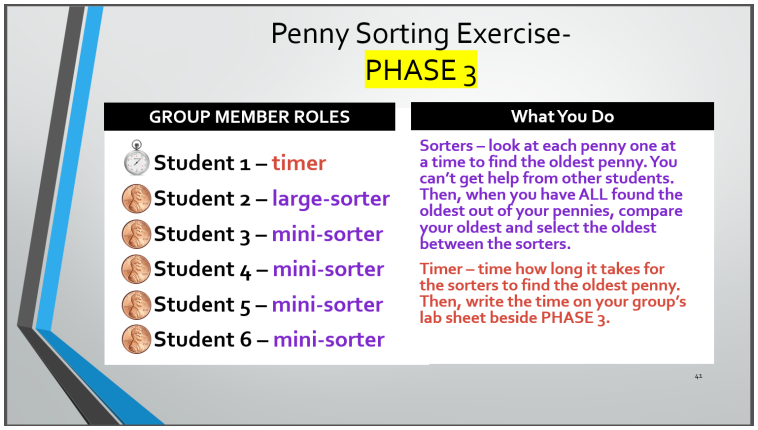


Figure 2.5: Penny Activity Phase 3

Phase 4 is shown in Figure 2.6. According to the lab instructors, students typically have the most fun with this phase. After the students have found the oldest penny in the room, the instructors talk about how this phase represents data parallelism as well, where the dataset (pennies) is divided among multiple workers (each worker only has one penny). We point out that ~~W~~with the data being spread across so many workers, the synchronization step where results need to be merged ends up taking more time than the actual work, and this affects speed-up.

2.3.1.1 Instructional Material (Lectures, Assignments)

Instructor materials for TNTECH's Unplugged Penny Activity can be found at here.

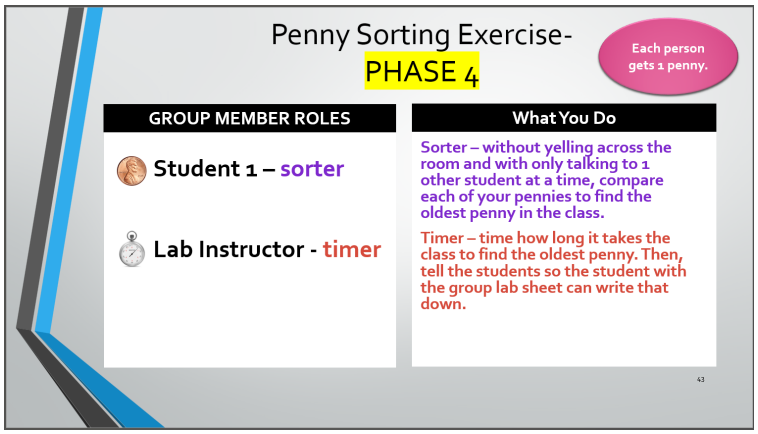


Figure 2.6: Penny Activity Phase 4

2.3.1.2 Challenges

We Accidentally Looked Like Coin Criminals. The Unplugged Penny Activity requires many pennies if you have a substantial number of students. Going to the bank and getting 50 dollars of pennies was a much bigger challenge than we thought it would be. First, the bank workers were very suspicious why we would make such a request. The bank teller behind the counter asked me why we needed that many pennies and told us that we would need to wait 15 to 20 minutes because they would have to be retrieved from the vault. Then, a bank manager came out and asked us the same question. While in the waiting room we felt we were being watched very closely. Then, they brought out a box that was small, but it weighed approximately 30 pounds (13.6 kilograms) and so it was much, much heavier than we anticipated. Figure 2.7 shows the heavy box of pennies that led to a heavy workout by loading it on the instructor's vehicle and then to their third floor office.



Figure 2.7: Box of 5,000 Pennies

Activity Could be Time Consuming. Because this is the first lab session of the semester, students are introduced to their lab instructors, view an "Introduction to Lab" presentation, complete the in-lab Penny Search Unplugged Activity, and verify that their

computers are properly configured to write, compile, and run C++ programs. The unplugged activity requires approximately 30–40 minutes of the 75-minute lab period. As a result, teaching assistants have reported that some students are unable to complete the computer setup process during class. To address this issue, setup instructions are distributed approximately one week before the semester begins, and multiple office-hour sessions are offered throughout the week to assist students who encounter difficulties. Also, some of the lab instructors combined Phase 1 & 2 into a single phase where they made it a competition and half of the groups would be doing Phase 1 whereas the other half would be doing Phase 2 at the same time. This helped reduce the time it took to complete this activity.

2.3.1.3 Data and Analysis

Figure 2.8 shows the results to two statements that we asked students to rate their level of agreement given a 7-point Likert scale. These responses refer to the whole lab class including being introduced to lab & lab instructors, doing the unplugged penny activity, and getting their computer set up for the class. Students were also asked, "What is something that you learned from this assignment?" and while the majority of responses were related to getting their computer set up or compiling & running their first program, there were six students who wrote "parallel computing."

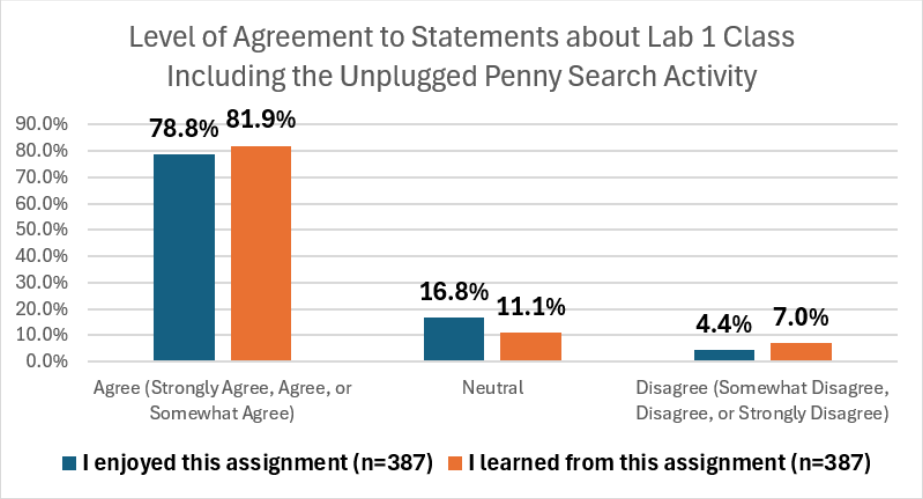


Figure 2.8: Level of Agreement for all of Lab 1 that included Unplugged Penny Activity

2.3.2 Unplugged Flag Maker Activity

Our data parallelism lab in week 11 (out of 16 weeks) begins with the Unplugged Flag Maker Activity, introduced in §1.3.1, and then students work on a plugged-in activity detailed in §§2.4. The Flag Maker Activity has four scenarios, but instructors can choose to do as many of the scenarios as time permits. The scenarios we used are explained below.

An image of all four scenarios is in Figure 1.2.

Students begin with **Scenario 1**. After the group is done with Scenario 1, they are instructed to write down the time it took to complete. In the first semester of doing this

activity, we had students write their times on a lab sheet where each group had a single lab sheet each. However, starting with the second semester, we started having students write their times up on a white board so that all students could see the times and could compare. This made the activity feel more like a competition or game. We also started handing out small prizes to winning (fastest) groups such as mini resin animals (approx. $10 for 86), flinging chickens (approx. $10 for 42), flinging ninjas (approx. $12 for 40), candy, mini feet (approx. $18 for 40), small squishy toys (approx. $22 for 130), fruit rollerball pens (approx. $9 for 12) and mini hands (approx. $10 for 30), which were purchased from Amazon.com. The mini hands, candy, and mini resin ducks were student's favorite prizes. We found that having students compete keeps all students engaged in the activity. The instructors then talk about how Scenario 1 represents a single processor where every action is sequential.

Then, students start **Scenario 2**. After recording times and distributing prizes for Scenario 2, the lab instructors ask the students to compare their times for Scenario 1 versus Scenario 2. We then explain that adding a second processor allows for parallel processing, so two stripes of the flag can be worked on simultaneously, cutting down completion time. We also discuss that this demonstrates how distributing tasks between processors can increase efficiency and speedup. We define speedup as the ratio of the time taken to solve a problem on a single processor to the time taken on a parallel system.

Next, students perform **Scenario 3**. During the discussion part of Scenario 3, lab instructors are able to point out that adding additional processors speeds up the work even more and they introduce the importance of synchronization between the processors. We talk about how if one processor finishes early, it sits idle, and how that is an example of underutilized resources. This allows the instructors to talk about how in real systems, load balancing is essential to ensure each processor has roughly the same amount of work to prevent idle time.

Last, students complete **Scenario 4**. During the discussion part of Scenario 4, lab instructors ask the following questions:

1. Was Scenario 4 faster or slower than Scenario 3?
2. Did each person (processor) use a marker, and then pass it to the next person to use – over and over until the work was done?
3. Was each person (processor) sometimes waiting for a marker?

The second question allows us to introduce pipelining as the technique of overlapping the execution of multiple instructions to improve performance. The third question allows us to talk about how when multiple processors are waiting on (competing for) shared resources, this is contention. We also talk about how students hands run into each other during this Scenario and how that represents race conditions - where processors might risk stepping into each other's workspaces at the boundaries.

2.3.2.1 Instructional Material (Lectures, Assignments)

Instructor materials for the Unplugged Flag Maker Activity can be found here.

2.3.2.2 Experience and Teaching Tips

The first semester that we did the Flag Maker unplugged activity, many students complained about having to use crayons because they broke easily and their hands hurt by the end of it. We changed over to markers the next semester and there were far fewer complaints.

The CS1 instructors meet weekly with the CS1 lab instructors (undergraduate teaching assistants) to perform a retrospective of the previous week's lab. We discuss what went well, what could be improved, and if there are any questions to be answered or problems to be resolved. In the Flag Maker retrospective meeting, the lab instructors said that in most labs, students enjoyed the activity except for in labs with smaller numbers of students. For example, a 20 to 30-person lab class enjoyed the activity more than the 10 to 15-person lab class. The TAs felt this was due to lack of being able to make it as much of a competition with fewer students. One TA mentioned that as long as the instructor makes it fun and engaging, students will feed off of the instructor's energy.

2.3.2.3 Data and Analysis

In Fall 2024, Spring 2025, Fall 2025, and Spring 2026 semesters, students took a Flag Maker pre-quiz that had multiple-choice and true/false questions and a post-quiz with these same questions as well as a survey. Figure 2.9 shows each question on the pre and post quizzes, how many students answered that question out of the four semesters, how many students got the question correct, and the percentage of students who answered correctly. The figure also shows the percentage difference between the pre and post quizzes. All but two questions resulted in a larger number of correct answers in the post-quiz.

Students also filled out a Flag Maker post-activity survey where they indicated their agreement with several statements. Figure 2.10 shows the results of the survey. The largest number of students (over 90%) agreed with "I put effort into the activity." and "The instructor(s) seemed prepared for the activity." Almost 80% of students agreed that they had fun during the activity and almost 72% agreed that the activity increased their understanding of parallel computing.

Qualitative thematic responses from the open-ended question "What is the most interesting thing you learned from the activity?" in the post-activity survey is shown in Table 2.9. Students also shared what they felt could be improved in the activity and the thematic responses from this question can be found in Table 2.10.

2.4 New Plugged-in Activities

2.4.1 Plugged-In Earthquake Activity

2.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

Details on this assignment are in §1.4.3. At TNTECH, we cover these topics in week 7 & 8 (out of 16) and then students typically are assigned this programming assignment during week 8 of the semester. They have three weeks to complete the assignment.

Flag Maker Pre-Quiz and Post-Quiz Question Text	Pre-Quiz Num. Answered (n)	Pre-Quiz Correct	Pre Percent Correct	Post-Quiz Num. Answered (n)	Post-Quiz Correct	Post Percent Correct	Change in Correct Count	Percentage-Point Change
What is a potential drawback of using multiple processors?	306	128	41.8%	308	258	83.8%	130	42.0%
What is a race condition in the context of parallel computing?	306	86	28.1%	311	165	53.1%	79	25.0%
What is contention in parallel computing?	600	321	53.5%	608	407	66.9%	86	13.4%
Speedup is defined as the ratio of the time taken to solve a problem on a single processor to the time taken on a parallel system.	599	464	77.5%	608	537	88.3%	73	10.8%
What is the primary limitation of using only one processor to complete a task?	308	281	91.2%	311	303	97.4%	22	6.2%
Scalability refers to the ability of a parallel system to increase its performance proportionally with the addition of more processors.	598	546	91.3%	604	567	93.9%	21	2.6%
What is the term for ensuring that each processor has roughly the same amount of work to prevent idle time?	307	204	66.4%	311	204	65.6%	0	-0.8%
Which of the following best describes task decomposition?	292	275	94.2%	301	277	92.0%	2	-2.2%

Figure 2.9: Results of Flag Maker Pre-Quiz and Post-Quiz

Table 2.9: Flag Maker Thematic Response to Student Interest in Activity

Interpretation of response to "*What is the most interesting thing you learned from the activity?*"	Responses	Percent of Responses (n=361)
Parallel processing, multicore processing, or simultaneous work	86	23.8%
Coloring, crayons, flag paper, etc.	76	21.1%
Speedup and efficiency benefits	34	9.4%
Additional processors can provide less benefit or even slow the task	30	8.3%
Task decomposition and load balancing	26	7.2%
Contention and shared-resource interference	25	6.9%
Synchronization, pipelining, race conditions	24	6.6%
Hands-on visualization	14	3.9%
Teamwork and collaboration	10	2.8%

Table 2.10: Flag Maker Thematic Response to Student Feedback in Activity

Interpretation of response to "*How could we improve the activity for future classes?*"	Responses	Percent of Responses (n=361)
Unsure or no suggestions	76	21.1%
No change needed or positive evaluation	52	14.4%
More explanation for new vocabulary or computing connection	43	11.9%
Better crayons, markers, or other materials	38	10.5%
Grouping, preparation, and organization	19	5.3%
Larger paper or better workspace/layout	18	5.0%
Shorter or less repetitive activity	15	4.2%]

Flag Maker Post-Activity Survey Likert Statements (*n=612*)	Percentage Agree	Percentage Neutral	Percentage Disagree
I had fun during the activity.	79.4%	13.6%	7.0%
I am confident in my understanding of the material presented during the activity.	75.3%	16.8%	7.8%
The activity increased my understanding of parallel computing.	71.6%	17.0%	11.4%
The activity made me more interested in parallel computing.	53.8%	34.6%	11.6%
The activity increased my understanding of loops.	40.8%	28.9%	30.2%
I was focused during the activity.	83.8%	10.3%	5.9%
I put effort into the activity.	91.0%	5.4%	3.6%
Explaining the material to my group improved my understanding of it.	59.8%	33.3%	6.9%
Having the material explained to me by my group members improved my understanding of it.	55.7%	36.1%	8.2%
Group discussion during the activity contributed to my understanding of parallel computing.	64.1%	25.7%	10.3%
Overall, the other members of my group made valuable contributions during the activity.	83.8%	12.4%	3.8%
I would prefer to take a class that includes this group activity over one that does not.	61.1%	26.1%	12.7%
I made a valuable contribution to my group during the activity.	85.0%	12.4%	2.6%
The instructor(s) seemed prepared for the activity.	90.2%	5.6%	4.2%
The instructor(s) put a good deal of effort into my learning from the activity.	86.9%	9.2%	3.9%
The instructor(s) enthusiasm made me more interested in the activity.	81.9%	13.1%	5.1%
The instructor(s) were available to answer questions during the activity.	88.9%	8.7%	2.5%

Figure 2.10: Results of Flag Maker Post-Activity Survey

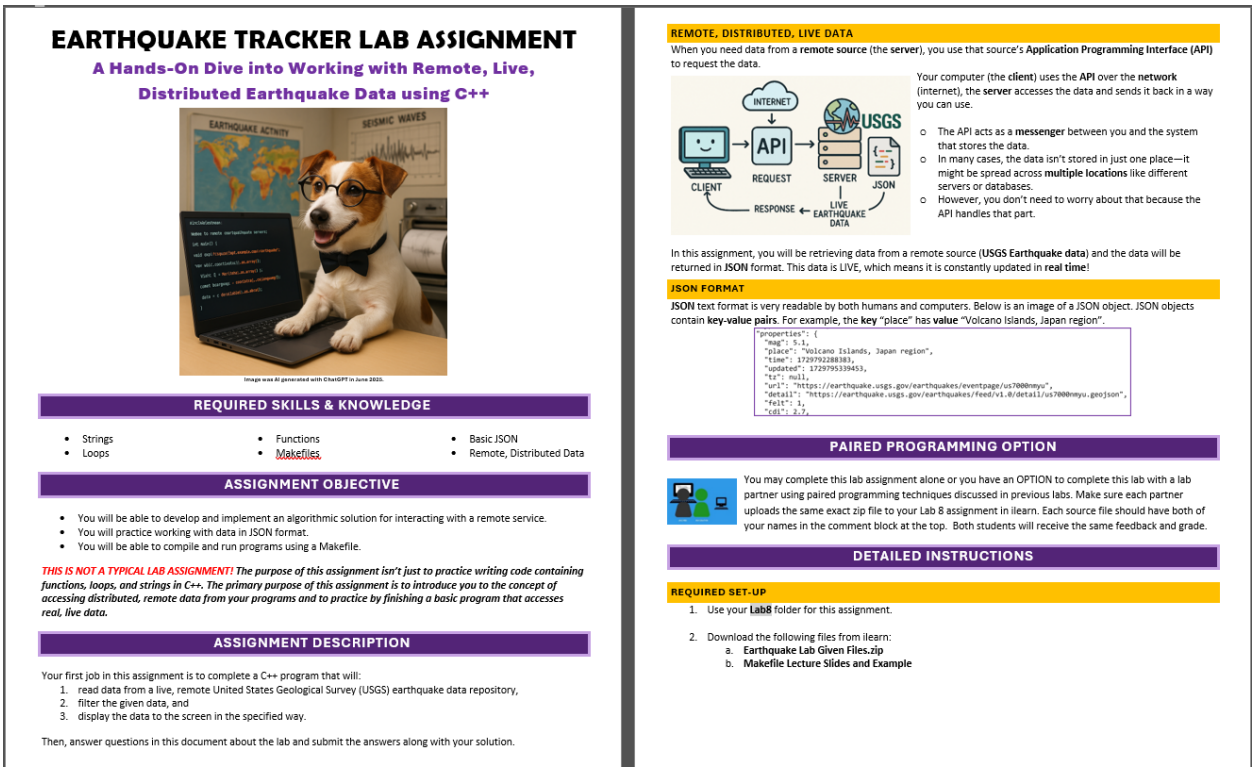
The **learning outcome** for this activity is that students will be able to develop and implement algorithmic solutions for interacting with a remote service.

2.4.1.2 Implementation Details

Students begin with a provided starter file, `Prog3_given.cpp`, which they rename to `Prog3.cpp`. They also receive supporting folders and files, including cURL files, a JSON library folder, Mac and Windows Makefile templates, and platform-specific setup instructions. The required setup asks students to place the correct Makefile directly in the Program 3 folder and, for Windows users, copy `libcurl-x64.dll` into the same folder.

The implementation uses several libraries beyond the standard introductory set. Students add `curl/curl.h` so the program can transfer data from a URL, `nlohmann/json.hpp`, so the program can access and read JSON data, and `sstream` so downloaded website data can be temporarily stored in a string stream. The assignment provides these libraries and explains their purpose so students can use them without needing to fully master external library development.

A key implementation requirement is that students should not modify the provided `downloadDataFromURL()` function. Instead, they should read the comment block and function body to understand its purpose, then use it as a helper function that returns the remote earthquake data. The `main()` function already contains some exception-handling code, including try and catch, which is not normally part of the CS1 curriculum, but students



EARTHQUAKE TRACKER LAB ASSIGNMENT

A Hands-On Dive into Working with Remote, Live, Distributed Earthquake Data using C++

REQUIRED SKILLS & KNOWLEDGE

- Strings
- Loops
- Functions
- Makefiles
- Basic JSON
- Remote, Distributed Data

ASSIGNMENT OBJECTIVE

- You will be able to develop and implement an algorithmic solution for interacting with a remote service.
- You will practice working with data in JSON format.
- You will be able to compile and run programs using a Makefile.

THIS IS NOT A TYPICAL LAB ASSIGNMENT! The purpose of this assignment isn't just to practice writing code containing functions, loops, and strings in C++. The primary purpose of this assignment is to introduce you to the concept of accessing distributed, remote data from your programs and to practice by finishing a basic program that accesses real, live data.

ASSIGNMENT DESCRIPTION

Your first job in this assignment is to complete a C++ program that will:

1. read data from a live, remote United States Geological Survey (USGS) earthquake data repository,
2. filter the given data, and
3. display the data to the screen in the specified way.

Then, answer questions in this document about the lab and submit the answers along with your solution.

REMOTE, DISTRIBUTED, LIVE DATA

When you need data from a **remote source** (the **server**), you use that source's **Application Programming Interface (API)** to request the data.

Your computer (the **client**) uses the **API** over the **network** (internet), the **server** accesses the data and sends it back in a way you can use.

- The API acts as a **messenger** between you and the system that stores the data.
- In many cases, the data isn't stored in just one place—it might be spread across **multiple locations** like different servers or databases.
- However, you don't need to worry about that because the API handles that part.

In this assignment, you will be retrieving data from a remote source (**USGS Earthquake data**) and the data will be returned in **JSON** format. This data is LIVE, which means it is constantly updated in **real time**!

JSON FORMAT

JSON text format is very readable by both humans and computers. Below is an image of a JSON object. JSON objects contain **key-value pairs**. For example, the **key** "place" has **value** "Volcano Islands, Japan region".

PAIRED PROGRAMMING OPTION

You may complete this lab assignment alone or you have an OPTION to complete this lab with a lab partner using paired programming techniques discussed in previous labs. Make sure each partner uploads the same exact zip file to your Lab 8 assignment in ilearn. Each source file should have both of your names in the comment block at the top. Both students will receive the same feedback and grade.

DETAILED INSTRUCTIONS

REQUIRED SET-UP

1. Use your Lab8 folder for this assignment.
2. Download the following files from ilearn:
   a. **Earthquake Lab Given Files.zip**
   b. **Makefile Lecture Slides and Example**

Figure 2.11: Screenshot of Earthquake Tracker Assignment

are told not to modify that portion. The inclusion of the exception-handling code allows students to encounter real-world C++ structure while still staying focused on the parts of the code aligned with our CS1 course learning goals for this module.

Compilation is also a major implementation component. Students do not compile this assignment using a simple one-line g++ command because the program depends on external files and library paths. Instead, they must update the provided Makefile with their local file paths, then use commands such as make, make run, and make clean. The rubric specifically evaluates whether the program compiles with no syntax errors or warnings using -Wall, runs correctly with the Makefile, and has a functional Makefile setup.

Students must submit `Prog3.cpp`, a `_documentation.docx` file, and proof that they completed the program report. The documentation file requires screenshots, timestamp conversion, short definitions of key distributed-data terms, and a real-world example of remote data use. The rubric gives 15 points for this documentation file and 5 points for program report proof, making reflection and explanation part of the overall assignment grade.

2.4.1.3 Instructional Material (Lectures, Assignments)

- Lecture Materials on Functions, Distributed Computing, JSON, cURL, Makefiles
- Assignment Document and Files Given to Students

2.4.1.4 Data and Analysis

We had students fill out a program report after each programming assignment and afterwards, they would upload a screenshot of the confirmation page along with their submission, which was part of their grade. In this program report, we asked students to estimate how long they spent on the program and when they started on the program. Figure 2.12 shows how much time students thought they spent on the assignment reported from students in Fall 2024 semester through Spring 2026 semester. Figure 2.13 shows how much time students felt they spent on the assignment just in Spring 2026 disaggregated by student's prior experience in programming before taking CS1.

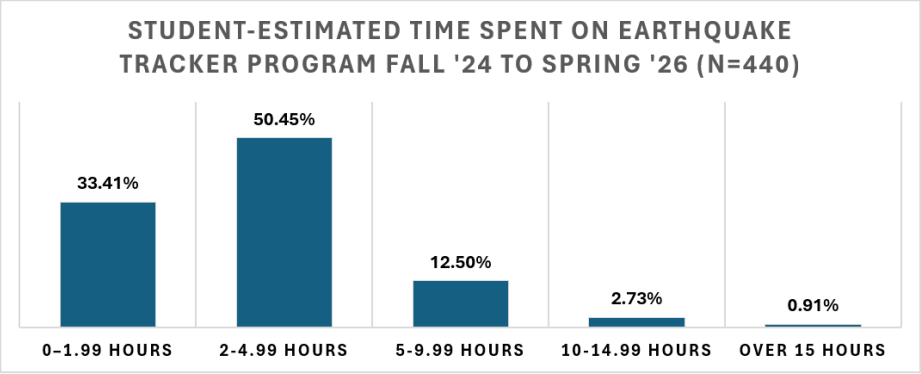


Figure 2.12: Time to Complete Earthquake Assignment (All Semesters)

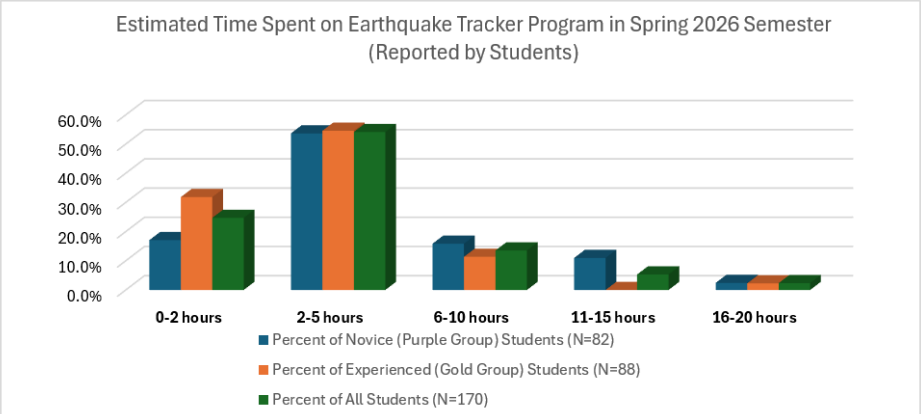


Figure 2.13: Estimated Time to Complete Earthquake Tracker Assignment

We also asked students what they learned from the assignment. The strongest learning outcome was that students saw how a C++ program can interact with remote, live, real-world data. Many responses specifically mentioned pulling data from a website, accessing a server, using an API, or retrieving live information. This suggests that the assignment successfully exposed students to a more authentic use of programming beyond isolated console input/output. A second major learning area was Makefiles and compilation. This is

interesting because Makefiles were probably not the conceptual centerpiece of the assignment, but they became highly salient to students. Many students said they learned how to use or run a Makefile, while others expressed frustration with them. This suggests that the build/setup process was a major part of the student experience. A third major theme was JSON. Students frequently mentioned learning how JSON works, how it is structured, or how to extract information from a JSON file. Several students connected JSON with APIs and live data, which suggests that they were beginning to understand the larger workflow: retrieve remote data, parse the JSON, and display selected information.

The program report also asked students what the most challenging aspect was of the assignment. The most frequent difficulties involved using the Makefile, compiling and running the program from the terminal, configuring file paths, and resolving external library issues such as cURL headers or missing `.dll` files. Many students experienced the assignment as a lesson in getting the build environment to work, which wasn't the intention.

Figure 2.14 shows the results to three statements that we asked students to rate their level of agreement given a 7-point Likert scale.

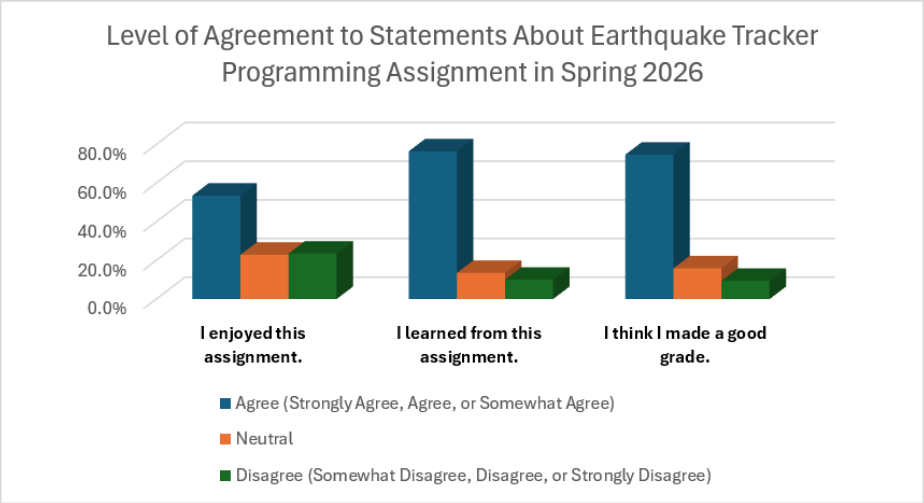


Figure 2.14: Level of Agreement Earthquake Tracker Assignment

2.4.1.5 Experience and Teaching Tips

Although many students found value in the assignment and recognized the relevance of APIs, JSON, remote data, and live real-world information, we feel that the assignment would be more enjoyable if there wasn't such a heavy lift for learning compilation, dependencies, and environment setup. For future iterations, we are going to consider adding short "tooling checkpoints" in lecture or lab before the full assignment, where students only verify that cURL, JSON support, and the Makefile work on their computer. That could reduce frustration and allow the main assignment to focus more clearly on the intended learning outcomes.

One teaching tip is to make sure to include requiring the documentation questions because they are pedagogically valuable. They give students a chance to explain the computing con-

cepts in plain language and connect the assignment to public-service systems beyond earthquakes, such as weather forecasting, emergency alerts, traffic navigation, disease tracking, or flight monitoring. These questions can help students recognize that APIs and distributed data are not abstract ideas; they are part of everyday systems people rely on.

2.4.2 Plugged-In OpenMP Data Parallelism Activity

2.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

This lab introduces students to data parallelism through a hands-on comparison between a sequential C++ program and an OpenMP-based parallel version. Students are given a completed sequential program, `sequential.cpp`, that reads strings from a text file into a vector, modifies each string by prepending its character count, sorts the vector alphabetically using bubble sort, and records the program's runtime. Students are also given an incomplete `parallel.cpp` file, which they must modify so that the data-processing portion of the program uses OpenMP parallelism. After implementing the parallel version, students run both programs on input files of increasing size and compare the execution times.

The purpose of the lab is not primarily to teach a new programming structure from scratch, but to help students observe how parallelism affects runtime, especially as data size increases. The lab emphasizes that parallel programming is increasingly important because modern computers rely on multi-core processors, and efficient software often needs to take advantage of that hardware. Students also see that parallelism is not automatically faster in all cases: for small input sizes, the overhead of creating and managing threads may make the parallel version slower, while larger input sizes may show a clearer benefit.

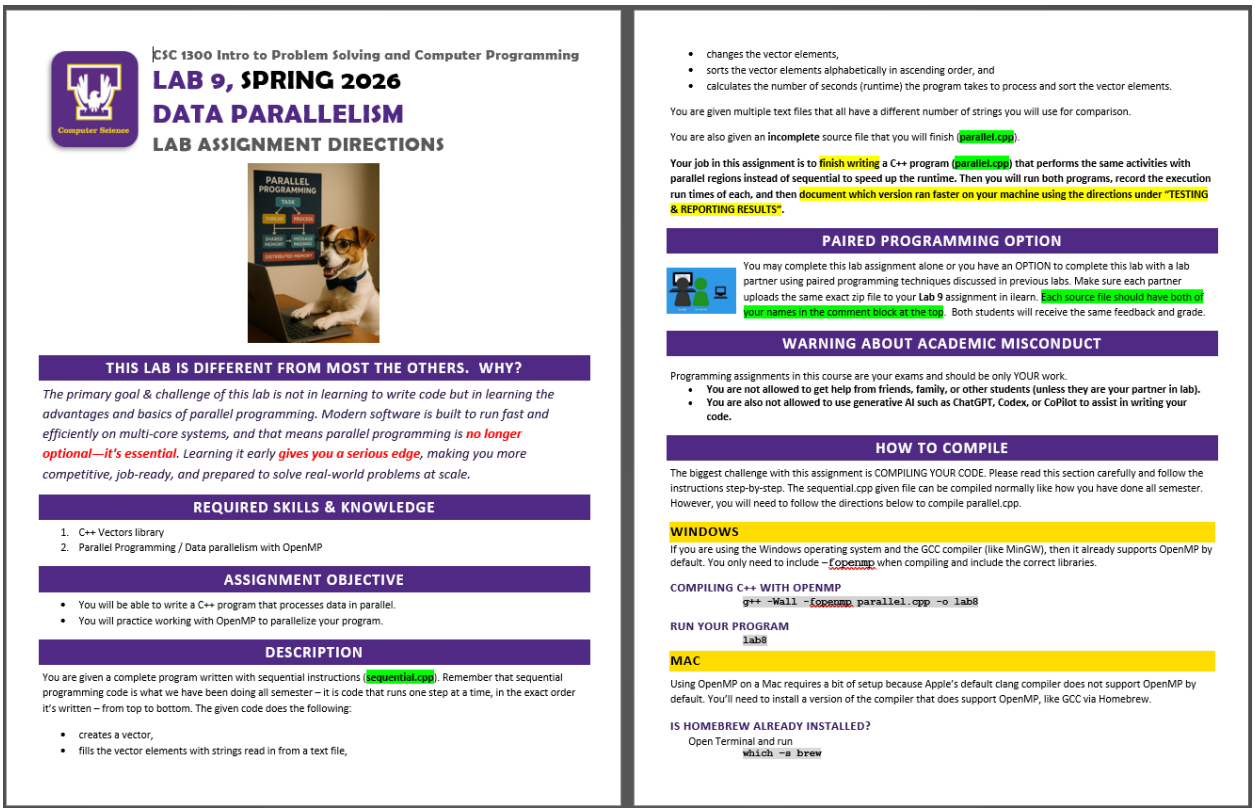
The **prerequisites** for this lab assignment include being comfortable with C++ vectors, loops, functions, file input, strings, and basic sorting logic. They also need introductory exposure to OpenMP concepts, including threads, parallel regions, and the `#pragma omp parallel for` directive. Because compiling OpenMP programs requires different commands than compiling standard C++ programs, students also need basic command-line compiling experience and some familiarity with their operating system's compiler setup.

Overall, the **learning outcome** for this activity is that students will have the ability to implement algorithmic solutions that are representative of data parallelism. Students should be able to explain the difference between sequential and parallel execution, use OpenMP to parallelize a loop, compile a C++ program with OpenMP support, collect runtime data across multiple input sizes, and interpret when parallelism improves performance and when it does not. They should also be able to connect their results to hardware factors such as the number of cores on their own machine.

2.4.2.2 Algorithms

The assignment centers on two related algorithms: a sequential data-processing algorithm and a parallelized version of that same algorithm.

In the sequential version, the program reads a list of strings from a text file and stores them in a vector. It then processes each string one at a time. For every string in the vector, the program counts the number of characters, converts that number to a string,



CSC 1300 Intro to Problem Solving and Computer Programming

LAB 9, SPRING 2026

DATA PARALLELISM

LAB ASSIGNMENT DIRECTIONS

THIS LAB IS DIFFERENT FROM MOST THE OTHERS. WHY?

The primary goal & challenge of this lab is not in learning to write code but in learning the advantages and basics of parallel programming. Modern software is built to run fast and efficiently on multi-core systems, and that means parallel programming is ***no longer optional—it's essential.*** *Learning it early* ***gives you a serious edge****, making you more competitive, job-ready, and prepared to solve real-world problems at scale.*

REQUIRED SKILLS & KNOWLEDGE

1. C++ Vectors library
2. Parallel Programming / Data parallelism with OpenMP

ASSIGNMENT OBJECTIVE

- You will be able to write a C++ program that processes data in parallel.
- You will practice working with OpenMP to parallelize your program.

DESCRIPTION

You are given a complete program written with sequential instructions (sequential.cpp). Remember that sequential programming code is what we have been doing all semester – it is code that runs one step at a time, in the exact order it's written – from top to bottom. The given code does the following:

- creates a vector,
- fills the vector elements with strings read in from a text file,
- changes the vector elements,
- sorts the vector elements alphabetically in ascending order, and
- calculates the number of seconds (runtime) the program takes to process and sort the vector elements.

You are given multiple text files that all have a different number of strings you will use for comparison.

You are also given an **incomplete** source file that you will finish (parallel.cpp).

Your job in this assignment is to finish writing a C++ program (parallel.cpp) that performs the same activities with parallel regions instead of sequential to speed up the runtime. Then you will run both programs, record the execution run times of each, and then document which version ran faster on your machine using the directions under "TESTING & REPORTING RESULTS".

PAIRED PROGRAMMING OPTION

You may complete this lab assignment alone or you have an OPTION to complete this lab with a lab partner using paired programming techniques discussed in previous labs. Make sure each partner uploads the same exact zip file to your **Lab 9** assignment in ilearn. Each source file should have both of your names in the comment block at the top. Both students will receive the same feedback and grade.

WARNING ABOUT ACADEMIC MISCONDUCT

Programming assignments in this course are your exams and should be only YOUR work.

- **You are not allowed to get help from friends, family, or other students (unless they are your partner in lab).**
- **You are also not allowed to use generative AI such as ChatGPT, Codex, or CoPilot to assist in writing your code.**

HOW TO COMPILE

The biggest challenge with this assignment is COMPILING YOUR CODE. Please read this section carefully and follow the instructions step-by-step. The sequential.cpp given file can be compiled normally like how you have done all semester. However, you will need to follow the directions below to compile parallel.cpp.

WINDOWS

If you are using the Windows operating system and the GCC compiler (like MinGW), then it already supports OpenMP by default. You only need to include –fopenmp when compiling and include the correct libraries.

COMPILING C++ WITH OPENMP

```
g++ -Wall -fopenmp parallel.cpp -o lab8
```

RUN YOUR PROGRAM

```
lab8
```

MAC

Using OpenMP on a Mac requires a bit of setup because Apple's default clang compiler does not support OpenMP by default. You'll need to install a version of the compiler that does support OpenMP, like GCC via Homebrew.

IS HOMEBREW ALREADY INSTALLED?

Open Terminal and run

```
which -a brew
```

Figure 2.15: Screenshot of Plugged-In OpenMP Data Parallelism Assignment

and prepends it to the original word using the format `count_word`. For example, apricot becomes `7_apricot`, and cat becomes `3_cat`. After every string has been modified, the program calls a bubble sort function to sort the vector in ascending alphabetical order. The program records the time required for the processing and sorting steps.

The parallel version preserves the same overall logic and output goal but changes how the string-processing loop is executed. Instead of processing each vector element one at a time in a strictly sequential loop, the program uses OpenMP to divide iterations of the loop among multiple threads. Because each string can be modified independently, this portion of the program is a good example of data parallelism: the same operation is applied to many independent elements in a collection. Students explicitly set the number of threads, then use an OpenMP directive before a traditional C++ for loop so that multiple elements can be processed at the same time.

The sorting step still uses bubble sort, which is intentionally simple but inefficient for large datasets. This helps make the cost of processing and sorting visible in the runtime results. Bubble sort repeatedly compares adjacent elements and swaps them when they are out of order. Although it is easy for beginning students to understand, it grows very slowly as the number of elements increases, making it useful for demonstrating how runtime changes as input size grows.

The lab also introduces an important performance concept: parallel speedup depends on both the amount of work being parallelized and the overhead required to manage threads. For small files, such as the file with 374 strings, the parallel version may be slower because

the overhead costs are larger than the benefit of dividing the work. For larger files, students are more likely to see the parallel version improve runtime.

2.4.2.3 Implementation Details

We prepare students for this assignment during a 55-minute lecture during week 9 after students have already been introduced to loops (during week 6) and arrays (during weeks 7 & 8). During this lecture, task parallelism and data parallelism is introduced and explained. Then, OpenMP is introduced and worked examples using OpenMP are provided and demonstrated. We first show a sequential example program running on a small file, then explain where the string-processing loop occurs. Next, we show how the same loop can be rewritten as a traditional index-based for loop and prepared for OpenMP.

Then, during the 75-minute lab class, this lab assignment is given to students. The assignment itself gives students several files of increasing size: words_374.txt, words_9330.txt, words_18660.txt, words_31100.txt, and words_466493.txt. These files allow students to observe how runtime changes as input size increases. Students were instructed to test correctness first with the smallest file, where printing the vector is still manageable, and then disable vector printing before collecting runtime data on the larger files.

2.4.2.4 Instructional Material (Lectures, Assignments)

- Lecture Materials on Arrays, Vectors, Parallelism, and OpenMP
- Assignment Document and Files Given to Students during the Lab

2.4.2.5 How-To Guide

Students need a clear guide for getting OpenMP set up in their particular computing environment and a guide for compiling OpenMP programs. The assignment contains separate Windows and Mac instructions because the setup differs substantially. On Windows with GCC or MinGW, students compile with the -fopenmp flag: `g++ -Wall -fopenmp parallel.cpp -o lab9` flag. On Mac, students may need to install Homebrew and then install GCC because Apple's default clang compiler does not support OpenMP by default. After installing GCC through Homebrew, students need to compile using a versioned compiler command and -fopenmp flag: `g++-14 -fopenmp -Wall parallel.cpp -o lab9`. (The exact version number may vary depending on the GCC version installed.)

Students also receive a short guide for checking the number of cores on their computer. This hardware information is needed to help teach students that runtime results are machine-dependent. Another useful guide provided in the assignment is a testing workflow. Students are instructed to first compile and run `sequential.cpp` to understand the baseline behavior. Then they are instructed to compile and run `parallel.cpp` with the smallest file, confirm that the output is correct, and only then move to larger files. Students are reminded to comment out or disable the `printVector()` function call before collecting timing results because printing a large vector will distort runtime measurements. They are also reminded to close unnecessary programs and avoid running heavy background tasks while timing their code.

Students are also given a simple results template along with the other given files. This template helps students focus on collecting and interpreting their data rather than guessing how to format the report.

2.4.2.6 Experience and Teaching Tips

This lab works best when students understand from the beginning that the goal is not simply to "make the code faster". The deeper goal is to investigate when and why parallelism helps. Some students may expect the parallel version to always be faster, so it is useful to explicitly discuss overhead before they begin. Thread creation, scheduling, and coordination all have costs. For small input sizes, those costs may outweigh the benefit of parallel execution. This makes the smaller input files pedagogically valuable because they help students confront a common misconception about parallel programming.

Compiling is likely to be the biggest practical obstacle, especially for Mac users. It is helpful to devote class or lab time to confirming that students can compile a basic OpenMP program before they work on the full assignment. They can use the provided program examples to test their computer. Students who cannot get OpenMP working may become stuck even if they understand the programming concept. A short "compiler check" activity before the lab can prevent many issues.

Students may also struggle with the difference between `for_each` and a traditional `for` loop. Since the starter code uses `for_each`, students need to understand why they are being asked to rewrite that portion. The clearest explanation is that OpenMP loop parallelization works naturally with standard index-based loops where the compiler can divide loop iterations among threads.

Another important teaching point is variable scope inside the parallel loop. Students should define the character-count variable inside the loop so that each thread has its own local copy. This helps avoid accidental sharing of a variable across threads and gives the instructor an opportunity to introduce the broader idea of shared versus private data in parallel programming.

It is also useful to frame the thread-count experiment as a discovery activity. When students change the number of threads to 2 and then to 500, they can see that more threads do not always mean better performance. Too few threads may not use the hardware effectively, while too many threads can create significant overhead. This gives students a more realistic understanding of performance tuning.

Finally, students should be reassured that their runtimes will not exactly match the sample output. Runtime depends on hardware, number of cores, operating system, compiler, background processes, and current system load. The most important outcome is not matching the instructor's times exactly, but collecting consistent results and explaining them thoughtfully.

2.4.2.7 Data and Analysis

As part of the lab assignment grade, students must fill out a lab report and upload the confirmation screenshot along with their submission to prove that they submitted the report. Students always have 6 days from the date of their lab class to submit assignments. Figure

2.16 shows student responses to asking "How many total hours did you spend on this lab assignment?"

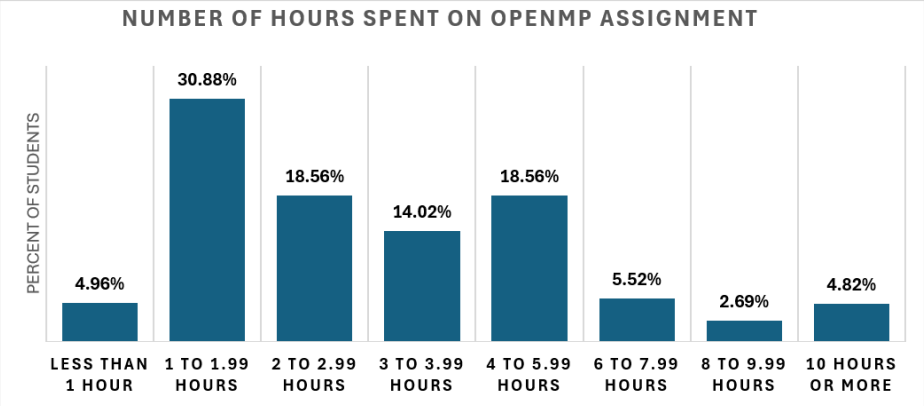


Figure 2.16: Hours Spent on Plugged-In OpenMP Lab Assignment

Figure 2.17 shows the common themes of responses from students when they were asked the open-ended question "What is something that you learned from this assignment?" The strongest theme was students' increased understanding of parallel computing. Nearly half of the respondents explicitly mentioned parallel programming, parallel processing, parallelism, or the difference between parallel and sequential execution. Responses ranged from basic recognition to more developed explanations of how work can be divided across multiple processing resources.

A particularly important learning outcome involved performance and efficiency. Many students recognized that parallel execution can reduce runtime, but they also learned that parallelization is not automatically faster. Students observed that performance depends on the task, the amount of data, the number of threads, and the computer's available cores. Several responses specifically noted that adding more threads does not always improve speed and can sometimes make a program slower. This suggests that students developed a more nuanced understanding than simply equating parallelism with improved performance.

Students also reported substantial learning related to arrays and data organization. These responses included using one- and two-dimensional arrays, iterating through arrays with loops, passing arrays to functions, and organizing information within a program. This indicates that the assignment reinforced foundational programming concepts while introducing parallel-computing ideas.

Below is a list of common challenges reported by students when they were asked the open-ended question "What was the most challenging aspect of the assignment? Is there anything you still have questions about?"

- Difficulty implementing parallel code or interpreting unexpected performance results
- Challenges creating, accessing, passing, or correctly indexing arrays
- Locating errors, correcting mistakes, and making output match expected results
- Difficulties defining functions, passing data, using references, and organizing files
- Problems compiling, configuring libraries, using run.sh, or working across operating systems

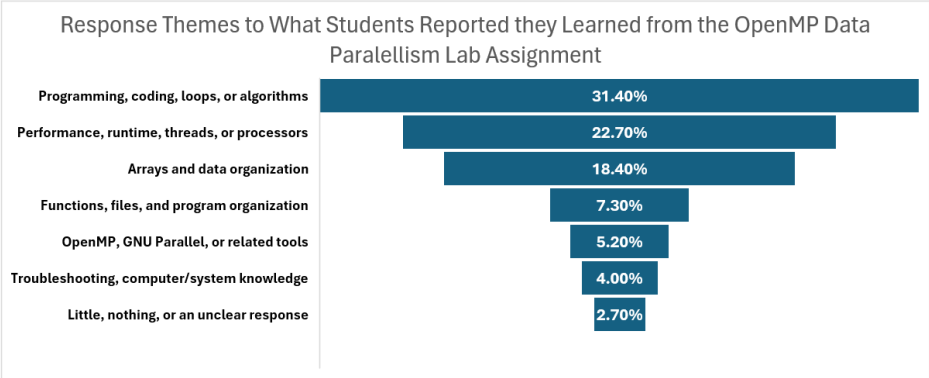


Figure 2.17: What Students Learned from Plugged-In OpenMP Data Parallelism Assignment

- Problems calculating values, processing files, and formatting or printing results
- Challenges writing loops, nested loops, parallel loops, and conditional statements

Students were also asked their level of agreement to two statements where students were provided a 7-point Likert scale. One statement was "I enjoyed this assignment" and the other was "I learned from this assignment." Figure 2.18 shows the results from students answering these questions.

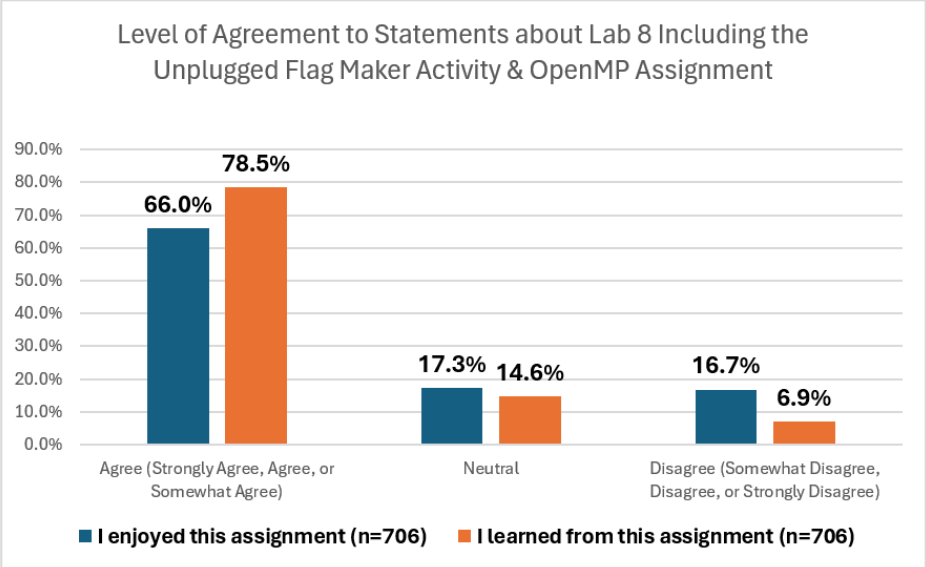


Figure 2.18: Level of Agreement OpenMP Data Parallelism Lab Assignment

2.5 Course-wide Pre and Post Course Surveys

To determine if our new and modified modules and assignments were effective at teaching basic PDC concepts while not reducing understanding of core, fundamental CS1 topics, we administered pre- and post-course surveys. Details about these surveys can be found in §1.5.

To download the actual surveys performed in CS1 and CS2 at TNTECH, go here. Table 2.11 shows how many students were enrolled in the CS1 course from Fall 2024 to Spring 2026. The table also shows how many consenting students filled out the Pre-Course survey and Post-Course survey each semester. These students' results are those reported in §2.5.1.

Table 2.11: Sample Size for Pre-Course and Post-Course Surveys

Population	Spr 2024	Fall 2024	Spr 2025	Fall 2025	Spr 2026	Total
No. Students Enrolled in CS1 Course	180	253	191	255	194	1256
No. Consented Students Pre-Course Survey	130	223	236	169	173	1090 (86.8%)
No. Consented Matched Pre-Post Surveys	127	166	123	196	140	879 (70%)

2.5.1 Results of Pre-Course and Post-Course Surveys

2.5.1.1 Spring 2024 CS1 Baseline Results

Table 2.12 summarizes the pre- and post-course survey results from the Spring 2024 benchmark semester, during which no formal PDC intervention was implemented. The *Pre* and *Post* columns present the mean student ratings collected at the beginning and end of the course, respectively. Responses were measured using a seven-point scale ranging from 1, indicating no experience at all, to 7, indicating an extremely high level of experience. Higher mean values indicate that students perceived themselves as having more experience with the concept, while lower values indicate little or no perceived experience. The *Diff* column represents the post-course mean minus the pre-course mean and can theoretically range from -6 to $+6$. A positive value indicates that students' average reported experience increased during the course, a value near zero indicates little change, and a negative value indicates that the average rating decreased. Larger positive differences represent greater pre-to-post growth. The *W* column reports the Wilcoxon signed-rank test statistic, which is calculated from the ranks of the differences between matched pre- and post-course responses. The value of W is interpreted together with the sample size and p-value; therefore, a high or low W value does not independently indicate whether the amount of improvement was large or important. The *p-value* column reports the probability of obtaining results at least as extreme as those observed if there were no systematic difference between the pre- and post-course responses. Values range from 0 to 1, with smaller values providing stronger evidence of a pre-to-post difference. In these analyses, results with a p-value below .05 were considered statistically significant. The *Sig.* column summarizes this decision as "Yes" when $p < .05$ and "No" when $p >= .05$. The *Cliff's* δ column provides an effect-size estimate ranging from -1 to $+1$. Because Cliff's delta was calculated in the pre-to-post direction, negative values indicate that post-course ratings generally tended to be higher than pre-course ratings, while positive values indicate that pre-course ratings tended to be higher. Values close to zero indicate little separation between the pre- and post-course responses, whereas values closer to either -1 or $+1$ indicate a stronger and more consistent difference.

The magnitude of the value, rather than its sign, is used to determine the size of the effect. The *Interp.* column categorizes that magnitude as negligible, small, medium, or large.

Table 2.12 shows that students reported statistically significant increases in their programming experience and in every traditional computing concept measured, with large effect sizes for data types, mathematical expressions, branching, loops, functions, reference variables, arrays, pointers, structures, and classes and objects. Among the PDC concepts, students reported a statistically significant but relatively small increase in experience with parallel processing, from 1.69 to 2.45. In contrast, experience with distributed computing remained essentially unchanged, while reported experience with event handling decreased from 2.65 to 1.74.

Table 2.12: Baseline Results: Spring 2024

Item	Pre	Post	Diff	*W*	*p*-value	Sig.	Cliff's δ	Interp.
Computing Background								
Computer Experience	5.54	5.66	0.12	2094.0	0.167	No	-0.04	negligible
Prog. Experience	3.44	4.58	1.14	463.0	0.000	Yes	-0.38	medium
Prog. Experience Compared to Classmates	3.23	4.39	1.16	735.0	0.000	Yes	-0.40	medium
Computing Concepts								
Data Types	3.57	6.52	2.95	0.0	0.000	Yes	-0.83	large
Math Expressions	3.57	6.41	2.84	0.0	0.000	Yes	-0.77	large
Branching	3.41	6.42	3.01	0.0	0.000	Yes	-0.81	large
Loops	4.07	6.43	2.36	23.0	0.000	Yes	-0.75	large
Functions	3.50	6.02	2.52	30.0	0.000	Yes	-0.74	large
Reference Variables	3.07	5.45	2.38	38.5	0.000	Yes	-0.66	large
Arrays	2.45	5.61	3.16	39.0	0.000	Yes	-0.81	large
Pointers	1.81	4.67	2.86	74.0	0.000	Yes	-0.77	large
C++ Structures	1.73	5.80	4.07	22.0	0.000	Yes	-0.90	large
Classes & Objects	2.43	4.84	2.41	158.0	0.000	Yes	-0.68	large
PDC Concepts								
Parallel Processing	1.69	2.45	0.76	534.0	0.000	Yes	-0.28	small
Distributed Computing	1.61	1.64	0.02	551.5	0.891	No	-0.01	negligible
Event Handling	2.65	1.74	-0.91	641.5	0.000	Yes	0.32	small

2.5.1.2 Fall 2024 through Spring 2026 CS1 Intervention Results

Table 2.13 presents the combined results from the Fall 2024 through Spring 2026 intervention semesters. Students reported significantly greater understanding across all traditional CS1 concepts, with large effect sizes, indicating that incorporating PDC material did not show evidence of reduced perceived progress of learning foundational programming knowledge. In addition, students reported significant gains in all three PDC areas: parallel processing increased from 1.83 to 3.77, distributed computing increased from 1.80 to 3.40, and event handling increased from 2.71 to 3.47. Parallel processing and distributed computing produced large effect sizes, while event handling produced a small effect size.

2.5.1.3 Conclusion of Results

The overall survey results suggest that students made significant pre-to-post perceived gains across both traditional CS1 topics and PDC-related concepts. After the PDC interven-

Table 2.13: Intervention Results: Fall 2024, Spring 2025, Fall 2025, and Spring 2026 Combined

Item	Pre	Post	Diff	W	p-value	Sig.	Cliff's δ	Interp.
Computing Background								
Computer Experience	5.54	5.65	0.11	29995.0	0.019	Yes	-0.03	negligible
Prog. Experience	3.41	4.66	1.26	6507.0	0.000	Yes	-0.44	medium
Prog. Experience Compared to Classmates	3.17	4.35	1.19	10330.5	0.000	Yes	-0.43	medium
Programming Languages								
C	1.30	1.70	0.40	325.0	0.000	Yes	-0.17	small
C++	1.70	4.66	2.96	80.5	0.000	Yes	-0.87	large
C#	1.28	1.52	0.23	247.0	0.000	Yes	-0.12	negligible
Java	1.78	1.84	0.06	1078.5	0.419	No	0.01	negligible
Javascript	1.86	1.86	-0.00	1546.5	0.866	No	0.01	negligible
PHP	1.11	1.12	0.01	68.0	1.000	No	-0.01	negligible
Python	2.72	2.78	0.06	3487.5	0.573	No	-0.01	negligible
Scratch	2.06	1.96	-0.10	1139.0	0.085	No	0.05	negligible
Visual Basic	1.19	1.30	0.11	185.0	0.133	No	-0.03	negligible
Other	1.90	1.89	-0.01	348.0	0.742	No	0.03	negligible
Computing Concepts								
Data Types	2.86	5.76	2.90	784.5	0.000	Yes	-0.73	large
Math Expressions	3.15	5.68	2.53	1334.5	0.000	Yes	-0.67	large
Branching	3.09	5.68	2.59	1132.5	0.000	Yes	-0.71	large
Loops	3.23	5.69	2.46	1098.0	0.000	Yes	-0.70	large
Functions	3.18	5.40	2.22	2356.5	0.000	Yes	-0.64	large
Reference Variables	2.83	4.91	2.07	4950.0	0.000	Yes	-0.62	large
Arrays	2.39	5.06	2.67	1511.0	0.000	Yes	-0.73	large
Pointers	1.63	4.28	2.66	787.5	0.000	Yes	-0.83	large
C++ Structures	1.43	4.68	3.25	339.0	0.000	Yes	-0.89	large
Classes & Objects	2.18	4.08	1.90	5188.5	0.000	Yes	-0.62	large
PDC Concepts								
Parallel Processing	1.83	3.77	1.94	8082.0	0.000	Yes	-0.68	large
Distributed Comp.	1.80	3.40	1.60	11266.0	0.000	Yes	-0.58	large
Event Handling	2.71	3.47	0.76	33627.0	0.000	Yes	-0.25	small
Computing Attitudes								
Interest in Computing	5.72	5.69	-0.03	26850.0	0.571	No	0.00	negligible
Interest in PDC	5.19	4.99	-0.20	35928.0	0.001	Yes	0.07	negligible
Importance of PDC	5.51	5.84	0.33	28566.0	0.000	Yes	-0.17	small
Understanding of Computing	4.22	5.28	1.05	13076.5	0.000	Yes	-0.37	medium
Understanding of PDC	3.13	4.75	1.63	9785.0	0.000	Yes	-0.56	large

tion, students continued to report strong growth in core programming knowledge, indicating that the added PDC material did not undermine foundational CS1 learning. For PDC topics, students showed statistically significant growth in parallel processing, distributed computing, and event handling. However, compared with the Spring 2024 benchmark, the size of the gains was similar rather than substantially larger. Parallel processing and distributed computing showed slight increases in mean gain after the intervention, while event handling showed a smaller gain, consistent with its limited instructional emphasis. Overall, the data supports the conclusion that PDC topics can be introduced into CS1 in a way that is feasible and non-disruptive, but larger gains in students' perceived PDC experience may require more explicit engagement with these concepts.

2.6 Conclusion

Parallel and distributed computing concepts are commonly reserved for upper-level courses and electives. However, the TNTECH CS1 exemplar demonstrates that introductory students can begin developing an awareness of parallel processing, distributed data access, event handling, and performance tradeoffs while they are still learning foundational programming concepts. Approximately four hours of PDC-related lecture and laboratory instruction were distributed throughout the semester and connected to existing course topics such as computer hardware, functions, arrays, vectors, file processing, classes, and graphical user interfaces.

The intervention did not require a fundamental restructuring of CS1. Instead, selected topics were condensed, existing assignments were modified, and several modular activities were added. Students continued to report substantial growth in traditional CS1 concepts while also reporting statistically significant growth in parallel processing, distributed computing, and event handling. These findings suggest that PDC concepts can be introduced in CS1 without displacing the programming knowledge students need for subsequent courses. The results should not be interpreted as demonstrating mastery of PDC. Rather, the intervention provided students with an early conceptual foundation and a more contemporary model of how software interacts with multicore hardware, remote data sources, external libraries, and asynchronous events.

- Lessons Learned
  - One important lesson was the need to design the assessment plan before collecting benchmark data. During the Spring 2024 benchmark semester, the post-course survey did not include questions about students' experience with individual programming languages, and the computing-attitude questions were not administered consistently. These items were added beginning with the first intervention semester because they provided useful context for interpreting students' growth. Future implementations should use the same measures during benchmark and intervention semesters whenever direct comparisons are intended.
  - A second lesson was that measures with high pre-course ratings may have limited room for improvement. Students entered the course already viewing PDC as important to computing. Consequently, changes in perceived importance were smaller than changes in students' perceived understanding and experience. Instructors and researchers should therefore avoid treating a small change in importance as evidence that the intervention was ineffective. Measures of knowledge, experience, understanding, and ability to apply concepts may be more sensitive indicators when students already recognize a topic's relevance.
  - A third lesson was that PDC is most accessible to beginning students when it is connected to concepts they are already learning. Parallelism was introduced with multicore hardware and arrays, distributed computing with functions and remote data, and event handling with classes and graphical interfaces. This placement helped the new content reinforce rather than compete with established CS1 outcomes. The activities were also modular, allowing individual components to be revised, replaced, or adopted independently.
  - The unplugged activities showed the value of giving students an intuitive experi-

ence before presenting implementation details. The Penny and Flag Maker activities made concepts such as speedup, task decomposition, data parallelism, load imbalance, synchronization, contention, and diminishing returns visible through physical actions. The competitive structure of the Flag Maker activity increased participation, but its success depended on instructor enthusiasm, group size, clear organization, and an explicit explanation of how each physical action represented a computing concept.

  - The plugged-in activities revealed that infrastructure can become more difficult than the intended computing concept. In the Earthquake Tracker assignment, students frequently focused on Makefiles, file paths, cURL, external libraries, and operating-system differences. Similarly, OpenMP compiler configuration was a substantial barrier, particularly on macOS. These tools provided authentic experience, but excessive setup difficulty could obscure the intended learning outcomes. Tooling should therefore be introduced deliberately and scaffolded rather than treated as a minor prerequisite.
  - Finally, preparing the full instructional team was essential. Because undergraduate teaching assistants led the laboratory sections, they needed to understand both how to conduct the activities and why PDC concepts belonged in CS1. Early implementation showed that procedural training alone was insufficient. Teaching assistants were more prepared to support students when training included the pedagogical rationale, expected learning outcomes, common misconceptions, technical setup, and opportunities to practice the activities before class.

- Teaching Tips for Instructors
  - Introduce PDC concepts gradually and connect each concept to a familiar CS1 topic. Students do not need to master parallel programming, distributed systems, or event-driven architecture in CS1. The objective is to establish accurate introductory mental models that can be revisited in later courses. Terminology should be limited, clearly defined, and repeatedly connected to concrete examples.
  - Use an unplugged or visual activity before asking students to modify parallel code. After each phase of an activity, pause to make the computing analogy explicit. Students may enjoy searching through pennies or coloring flags without independently recognizing the relationship to processors, threads, synchronization, load balancing, or contention. Guided reflection questions are therefore a necessary part of the activity rather than an optional conclusion.
  - For the Flag Maker activity, display group completion times where everyone can see them and use the results to discuss speedup and performance variation. Small prizes can make the activity more engaging, but the competition should remain low stakes. In smaller lab sections, instructors may need to adapt the activity by comparing multiple trials, competing against instructor-provided benchmark times, or combining groups so that the activity retains sufficient energy.
  - Plan the Penny Activity carefully when it is included in the first laboratory meeting. At TNTECH, the activity required approximately 30–40 minutes of a 75-minute lab and competed with introductions and computer setup. Distributing setup instructions before the semester, offering setup assistance outside class,

reducing the number of phases, or having groups perform different phases concurrently can preserve the activity without preventing students from completing required software configuration. Instructors may also substitute lighter and more portable objects for pennies when storage, transportation, accessibility, or cost is a concern.

  - Conduct tooling checkpoints before assigning the Earthquake Tracker or OpenMP activities. Students should verify that they can compile a small program using the required compiler, libraries, and flags before beginning the full assignment. Separate instructions should be provided for supported operating systems, and instructors should verify those instructions immediately before each semester because compiler versions, library paths, and installation procedures may change.
  - Keep starter code focused on the intended learning outcomes. Complex supporting operations, such as downloading data, handling exceptions, or configuring libraries, can be provided as completed functions that students inspect and call without modifying. Clearly distinguish between code students are expected to understand conceptually, code they must modify, and infrastructure code they may treat as a provided abstraction.
  - When teaching the OpenMP activity, emphasize that parallel execution is not automatically faster. Small datasets, excessive thread counts, thread-management overhead, hardware differences, and background processes can all affect runtime. Students should be evaluated on whether they collect data systematically and explain their results rather than on whether their runtime matches an instructor-provided value.
  - Include short written reflection or documentation questions in programming assignments. Asking students to define concepts in their own words, interpret timing results, and identify real-world applications helps distinguish conceptual learning from merely producing working code. These reflections are particularly valuable when starter code or external libraries hide some of the underlying implementation complexity.
  - Finally, provide teaching assistants with advance access to all materials and use a rehearsal-and-retrospective cycle. Before each activity, teaching assistants should complete the activity, test the software setup, discuss expected misconceptions, and practice the conceptual explanation. Afterward, the instructional team should document problems and revise timing, grouping, instructions, and examples for the next offering.

- Inferences and Limitations
  - The combined results indicate that PDC-related material can be infused into a large, C++-based CS1 course without preventing students from reporting strong growth in foundational programming concepts. The gains in parallel processing and distributed computing also suggest that beginning students can develop meaningful introductory experience with these concepts when they are reinforced through multiple forms of instruction. However, these findings should be interpreted as evidence of feasibility and perceived learning rather than proof of PDC mastery.

  - Several limitations constrain the conclusions that can be drawn. First, much of the course-wide evidence was based on self-reported experience and understanding. Student perceptions provide useful information about confidence and familiarity, but they do not directly measure students' ability to independently implement or transfer PDC concepts. Activity-level quizzes, assignments, and reflections provide additional evidence, but they were not equivalent to a comprehensive objective assessment of PDC knowledge.
  - Second, the study used a benchmark-and-intervention design rather than randomly assigning students to PDC and non-PDC versions of the course. The benchmark and intervention groups were drawn from different semesters and may have differed in prior experience, academic preparation, enrollment patterns, instructor or teaching-assistant composition, and other contextual factors. Therefore, observed differences should not be interpreted as demonstrating that the intervention alone caused the reported outcomes.
  - Third, the benchmark survey did not contain every item included in later intervention surveys. In particular, programming-language and computing-attitude measures were added after the benchmark semester. Direct benchmark-to-intervention comparisons are therefore not possible for all outcomes. Additionally, the course-wide analysis included only students who consented and completed matched pre- and post-course surveys. Students without matched responses may have differed systematically from those represented in the results.
- Future Work
  - Future work at TNTECH will build on the PDC-infused CS1 exemplar by investigating how additional modern computing topics can be integrated into the introductory curriculum without weakening foundational programming outcomes. A primary direction is the infusion of BigData and AI concepts into CS1. As with the PDC intervention, these topics will be connected to existing course content rather than introduced as isolated advanced units.
  - Big Data activities may introduce students to the characteristics of large and varied datasets, limitations of local processing, data acquisition from remote sources, data cleaning, sampling, aggregation, and the relationship between dataset size and computational performance. The Earthquake Tracker assignment provides a natural starting point because students already retrieve and process live structured data. Future versions could allow students to compare datasets, summarize larger collections of records, identify patterns, and consider how the volume or velocity of data changes the computational approach.
  - Future evaluations should include direct assessments of conceptual knowledge and transfer in addition to self-reported experience. Students could be asked to select an appropriate sequential, parallel, distributed, event-driven, data-intensive, or AI-supported approach for a new problem and justify their choice. Semester-level analyses should also examine whether outcomes vary according to prior programming experience, instructional group, major, course performance, or other student characteristics.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter2-TTU

Detailed Pre and Post Syllabus

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter2-TTU/Detailed_Pre_and_Post_Syllabus

Organization of Material

The supporting materials for the TNTECH CS1 course package are available in the TNTECH CS1 GitHub repository. The repository is organized into the following categories:

- **Course Overview and Calendar**
  - `README.md`: Provides an overview of CSC 1300, including the institution, course level, programming language, included activities, and contact information.
  - `COURSE_CALENDAR.md`: Provides a general calendar identifying the placement of course topics and activities throughout the semester.
- **Detailed Post-Intervention Syllabus and Course Schedule**
  The `Detailed_Pre_and_Post_Syllabus` directory contains the detailed Spring 2026 course documents:
  - CSC 1300 Spring 2026 Syllabus
  - CSC 1300 Spring 2026 Master Schedule

  These files document the organization of the course after the integration of parallel, distributed, and event-driven computing topics.
- **Plugged-In Earthquake Tracker Assignment** The Plugged-In Earthquake Tracker Assignment directory contains an assignment in which students develop an earthquake-tracking program while exploring distributed computing, remote services, and APIs.
  - Activity overview and instructions
  - Earthquake Tracker assignment directions
  - Earthquake Tracker grading rubric
  - Module 7 Functions and Distributed Computing presentation slides
  - Earthquake Program Given Files: Contains the C++ starter program, JSON-support files, and platform-specific Makefiles supplied to students.
  - CSC 1300 Makefiles presentation
  - Mac Makefile Example: Contains a driver program, function definitions, a header file, and a Makefile configured for macOS.
  - Windows Makefile Example: Contains a driver program, function definitions, a header file, and a Makefile configured for Windows.

- **Unplugged and Supplemental Instructional Activities**
  The `Unplugged_Activities` directory contains hands-on activities and supplemental lecture materials.
  - **Unplugged Penny Search Activity** The Unplugged Penny Search Activity introduces search strategies, task distribution, and parallel work through a hands-on exercise.
    * Activity overview and implementation information
    * Penny Search Activity presentation
    * Penny Sorting Exercise
  - **Unplugged Flag Maker Activity** The Unplugged Flag Maker Activity introduces parallel programming and data parallelism through a collaborative flag-making exercise.
    * Activity overview and implementation information
    * Flag Maker Activity presentation slides
    * Printable Flag Maker scenario grids: Contains four printable scenario grids used during the activity.
  - **Event Handling Lecture** The Event Handling Lecture directory contains materials introducing object-oriented programming, graphical user interfaces, and event handling.
    * Lecture overview
    * Module 11 Object-Oriented Programming and Event Handling presentation
  - **zyBooks Parallel Computing Materials** The zyBooks Parallel Computing directory contains supplemental materials associated with the parallel-computing content delivered through zyBooks.
- **Evaluation Data and Survey Instruments**
  The `Evaluation_Data_and_Analysis/Survey_Instruments` directory contains informed-consent forms, course-experience surveys, activity quizzes, activity surveys, lab reports, and programming-assignment reports collected from Spring 2024 through Spring 2026.
  - **Spring 2024**
    * CS1 Spring 2024 Informed Consent Form
    * CS1 Spring 2024 Post-Course Experience Survey
  - **Fall 2024**
    * CS1 and CS2 Fall 2024 Informed Consent Form
    * CS1 Fall 2024 Pre-Course Experience Survey
    * CS1 Fall 2024 Post-Course Experience Survey
    * CS1 Fall 2024 Pre-Activity Flag Maker Quiz
    * CS1 Fall 2024 In-Lab Flag Maker Activity Survey
    * CS1 and CS2 Fall 2024 Lab Report
  - **Spring 2025**

        * CS1 and CS2 Spring 2025 Informed Consent Form
        * CS1 Spring 2025 Pre-Course Experience Survey
        * CS1 Spring 2025 Post-Course Experience Survey
        * CS1 Spring 2025 Pre-Activity Flag Maker Quiz
        * CS1 Spring 2025 In-Lab Flag Maker Activity Survey
        * CS1 and CS2 Spring 2025 Lab Report
    - **Fall 2025**
        * CS1 and CS2 Fall 2025 Informed Consent Form
        * CS1 Fall 2025 Pre-Course Experience Survey
        * CS1 Fall 2025 Post-Course Experience Survey
        * CS1 Fall 2025 Pre-Activity Flag Maker Quiz
        * CS1 Fall 2025 Post-Activity Flag Maker Quiz and Survey
        * CS1 and CS2 Fall 2025 Lab Report
        * CS1 and CS2 Fall 2025 Programming-Assignment Report
    - **Spring 2026**
        * CS1 and CS2 Spring 2026 Informed Consent Form
        * CS1 Spring 2026 Pre-Course Experience Survey
        * CS1 Spring 2026 Post-Course Experience Survey
        * CS1 Spring 2026 Pre-Activity Flag Maker Quiz
        * CS1 Spring 2026 Post-Activity Flag Maker Quiz and Survey
        * CS1 and CS2 Spring 2026 Lab Report
        * CS1 and CS2 Spring 2026 Programming-Assignment Report
- **Optional Archive of Syllabi and Course Schedules**
  The `Optional_Archive/Syllabi & Course Schedules` directory contains earlier syllabi and master schedules that support comparisons across implementation semesters.
    - CSC 1300 Fall 2024 Syllabus
    - CSC 1300 Fall 2024 Master Schedule
    - CSC 1300 Spring 2025 Syllabus
    - CSC 1300 Spring 2025 Master Schedule
    - CSC 1300 Fall 2025 Syllabus
    - CSC 1300 Fall 2025 Master Schedule

  The instruments include informed-consent forms, pre-course and post-course experience surveys, pre-activity and post-activity quizzes, activity surveys, lab reports, and programming-assignment reports collected between Spring 2024 and Spring 2026.
- **Optional Archive**
  The `Optional_Archive/Syllabi & Course Schedules` directory contains earlier syllabi and master course schedules:
    - CSC 1300 Fall 2024 Syllabus
    - CSC 1300 Fall 2024 Master Schedule
    - CSC 1300 Spring 2025 Syllabus

- CSC 1300 Spring 2025 Master Schedule
- CSC 1300 Fall 2025 Syllabus
- CSC 1300 Fall 2025 Master Schedule

These archived files allow instructors and researchers to compare course organization across the baseline and intervention semesters.

Together, these materials provide the instructional resources necessary to reproduce or adapt the TNTECH implementation, as well as the course documentation and evaluation instruments needed to examine how the intervention was incorporated and assessed.

Survey and Other Anonymized Data and Analysis

Data and analysis about the added activities can be found in the individual sections about the activities:

- Unplugged Penny Search Activity Data & Analysis 2.3.1.3
- Unplugged Flag Maker Activity Data & Analysis 2.3.2.3
- Plugged-In Earthquake Tracker Assignment Data & Analysis 2.4.1.4
- Plugged-In OpenMP Data Parallelism Assignment Data & Analysis 2.4.2.7

Section 2.5.1 contains results of pre-course and post-course surveys from the Spring 2024 CS1 Baseline semester and the combined Fall 2024 through Spring 2026 CS1 PDC Intervention semesters. https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter2-TTU/Evaluation_Data_and_Analysis/Survey_Instruments

Chapter 3

CS1 Course Package – Knox College

Java-Based Liberal-Arts Adaptation with Flag Maker, Greenfoot, and Knoxcraft[†]

David P. Bunde[1] and Jaime Spacco[2]

[1]Knox College, dbunde@knox.edu
[2]Knox College, jspacco@knox.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 108
3.1 Introduction . . . 110
3.2 How CS1 was changed and PDC topics integrated . . . 110
3.2.1 Course calendar . . . 110
3.3 New Activities . . . 111
3.3.1 Flag Maker . . . 111
3.3.1.1 Activity Summary . . . 111
3.3.1.2 Prerequisites . . . 111
3.3.1.3 Learning Outcomes . . . 111
3.3.1.4 Implementation Details . . . 111
3.3.1.5 Instructional Material . . . 112
3.3.1.6 Experience and Teaching Tips . . . 113
3.3.1.7 Data and Analysis . . . 113
3.3.2 Knoxcraft . . . 114
3.3.2.1 Activity Summary . . . 114
3.3.2.2 Prerequisites . . . 114
3.3.2.3 Learning Outcomes . . . 115
3.3.2.4 Implementation Details . . . 115
3.3.2.5 Instructional Material . . . 115
3.3.2.6 Experience and Teaching Tips . . . 115
3.3.2.7 Data and Analysis . . . 116
3.3.3 Greenfoot . . . 116
3.3.3.1 Activity Summary . . . 116
3.3.3.2 Prerequisites . . . 116
3.3.3.3 Learning Outcomes . . . 116
3.3.3.4 Implementation Details . . . 116
3.3.3.5 Instructional Material . . . 116
3.3.3.6 Experience and Teaching Tips . . . 116
3.3.3.7 Data and Analysis . . . 117
3.4 Course-wide Pre and Post Course Surveys . . . 117
3.5 Conclusion . . . 119
3.5.1 Experiences and Advice . . . 119
Appendix . . . 122

Chapter figures

3.1 Flag coloring assignment version of the flag of Great Britain . . . 112
3.2 Dependency graph for coloring the flag of Great Britain . . . 112
3.3 Flag coloring assignment version of the flag of Jordan . . . 113
3.4 Dependency graph for coloring the flag of Jordan . . . 114
3.5 Parallel Knoxcraft Turtles. . . . 115

Chapter tables

3.1 CS1 Pre/Post Survey Analysis Results (Winter 2024) . . . 117
3.2 CS1 Pre/Post Survey Analysis Results (Winter 2026) . . . 118
3.3 Post-Survey Course Resource and Activity Helpfulness Ratings (1–5 Scale) . . . 118

3.4 CS1 Pre/Post Computing Attitudes Analysis Results (Winter 2026) 119

Abstract

Knox College hosted the Java-based development teams in this project. Its CS1 course was updated with three new activities: Unplugged Flag Maker Activity, Knoxcraft, and using Greenfoot as a game engine (Greenfoot Activity). These activities introduce the ideas of parallel execution and dependencies, give students experience interacting with a remote server, and has them write event-driven code. Although these activities cover this new material, the focus of what students actually do is on traditional topics such as loops and object state so that they emerge from the class with the skills their peers had in previous terms, but with greater exposure to modern topics.

Descriptor Elements

Textbook used: None

Programming Language Employed: Java

Relevant core courses: CS1

Pre-requisites: None

Relevant PDC topics: Classification of PDC Topics according to Bloom's Taxonomy [1]: Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR)

	Chapter Section		
	Flag Maker	Knoxcraft	Greenfoot
PDC Concept	§3.3.1	§3.3.2	§3.3.3
Speedup	UN	UN (optional)	
Parallel problem decomposition	AN	AP (optional)	
Data parallelism	AP		
Dependencies	AP		
Distributed computing		UN	AP
Event handling			AP

Learning outcomes: After this course, in additional to the existing learning outcomes from the previous version of the course, students are able to do the following:

1. define and calculate speedup,
2. give examples of data parallel computing,
3. explain dependencies and give examples when they occur,
4. give examples when a remote server is useful for a program, and
5. write simple programs that use event handling.

Context for use: Knox College is a small, residential liberal arts institution. The academic calendar has 3 terms (Fall, Winter, and Spring), each slightly less than 10 weeks long. The CS1 class meets 4 times a week, 3 lectures (MWF) and one lab meeting per week. All class meetings are 70 minutes long. The intent is that a Knox course should have equivalent class time to a semester course at other institutions and cover the same amount of content.

The size of this class varies, but the targeted size is 24 students. The class is the first course in the Computer Science major, the Computer Science minor, and the Data Science major. It is also required for students majoring in Biology and Neuroscience who complete a BS instead of a BA in those fields. The course is also a common option for students in the Business & Management major and is taken by many students as an elective. Historically, roughly half of the students from this course continue to CS2.

This course uses Peer Instruction [47, 49], a pedagogy where lecture is periodically interrupted with a multiple-choice question. The students answer these individually, then discuss them in groups and re-answer. Then the instructor reveals the correct answer, has a student explain this answer, and then fills in any additional material. The in-class questions are graded only for participation. Lab assignments are homework; the students begin the assignments in lab, when the instructor and a TA are available to answer questions. The in class work gets them over the hurdle of starting the assignment. Then they have until the beginning of the next lab (a week) to complete the assignment.

3.1 Introduction

As one of the development teams, Knox introduced three significant changes to its CS1 course (Flag Maker, Knoxcraft, and Greenfoot) to introduce data parallelism, distributed computing, and event handling, the three topics identified as suitable for CS1 early in the project [12]. We also preserved other aspects of our modernization goals; the course already included graphical assignments (which we increased in number) and used modern pedagogy (Peer Instruction [47, 49], which we retained).

The next section (§3.2) discusses the overall changes. The bulk of this chapter the describes the new activities: Flag Maker (§3.3.1), Knoxcraft (§3.3.2), and Greenfoot (§3.3.3). Then we conclude the chapter (§3.5).

3.2 How CS1 was changed and PDC topics integrated

The time for the new material came out of a previously-assigned 2-week Media Computation lab in which students implement image processing operations by manipulating image pixels. (It was called Photoshop Jr.) The focus of this assignment was on loops, particularly nested loops. The course also covered nested loops in the programming part of Plugged-In Flag Maker Activity, which was already included. Thus, we added the unplugged and parallel parts of Flag Maker (roughly 1 lecture day). One of the weeks was then devoted to Knoxcraft, which also covers loops and nested loops. The other week was used to expand the existing unit on object-oriented programming into the Greenfoot Activity, which also includes this material (in a more compelling way) and adds exposure to remote servers and distributed computing.

3.2.1 Course calendar

Here is the new course calendar at a week-by-week level:

Week	Topics	Changes
1	Course introduction; types and variables; for loops; calling constructors and methods	
2	Conditionals; booleans; loops	
3	Strings; nested loops; data parallelism	Flag Maker §3.3.1, §1.3.1
4	Writing functions (static methods); return values	
5	Arrays; Arraylist; identifying mystery functions	
6	Effective comments; using a server	Knoxcraft §3.3.2
7	Writing classes; event-driven programming	Greenfoot §3.3.3
8	RPC; more practice	Greenfoot §3.3.3
9	Recursion	

Day-by-day level detail as well as the specific lecture materials and assignments are provided in the online materials accompanying this chapter, as detailed in the chapter's github repo.

3.3 New Activities

3.3.1 Flag Maker

3.3.1.1 Activity Summary

This activity uses the task of coloring the pixels of a flag to motivate students to practice for loops and then unplugged activities to introduce parallel concepts. It builds on the previously-described Plugged-In Flag Maker Activity (§1.4.1) and Unplugged Flag Maker Activity (Section 1.3.1) activities. After the unplugged activity, we then added some explicit discussion of parallel task decomposition and dependencies. The entire activity is described by Smith et al. [57]; some of the material here is taken from that description.

3.3.1.2 Prerequisites

We used this activity early in the term (week 3) so the prerequisites are minimal. Students do need to be able to write for loops and call methods for the programming aspects of the assignment. (They do get to follow a sample implementation.) The prerequisites for the unplugged components are even weaker; students need to have an understanding of the step-by-step nature of code. (Some of the slides after the unplugged activity explicitly reinforce students' ability to trace nested loops.)

3.3.1.3 Learning Outcomes

The learning goals for this activity are mainly the learning goal inherited from Unplugged Flag Maker Activity and Plugged-In Flag Maker Activity. At the end of this activity, students should be able to do the following:

1. implement various nested loop configurations to traverse linear, triangular, and rectangular regions of index-space;
2. recognize parallel computing terms such as speedup, contention, and pipelining;
3. recognize algorithmic solutions that are representative of data parallelism;
4. describe how parallel execution time is bounded by the slowest processor (i.e., the critical path);
5. be able split a simple problem into tasks and identify dependencies between them.

3.3.1.4 Implementation Details

In week 3 of the course, the students had already been introduced to integer variables and operations, boolean variables and conditional expressions, calling constructors and methods, and `for` loops. Then they are given the Plugged-In Flag Maker Activity (§1.4.1) in lab and

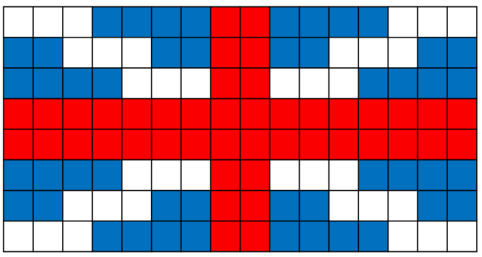


Figure 3.1: Flag coloring assignment version of the flag of Great Britain

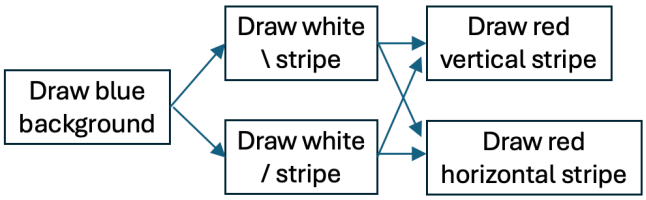


Figure 3.2: Dependency graph for coloring the flag of Great Britain

told to complete it as homework. The next day, we do the Unplugged Flag Maker Activity (§1.3.1).

At the end of that same period, we go through a deck of slides. The first couple of these reinforce loop concepts (which cell is visited first in a traversal by nested loop and which cell is visited second). Then the slides introduced the idea of dependencies. The coloring pixels in parallel is in tension with an important technique for more complicated flags: coloring different elements of the flag in layers. For example, the flag of Great Britain (Figure 3.1) is most easily made by coloring the entire flag blue, then adding the crossing diagonal white lines, and then finally coloring the red vertical and horizontal lines. This approach avoids having to make complicated intersection tests between the flag's different features. (The idea is the same as the Painter's algorithm in 3D graphics, which renders complex scenes by drawing polygons in the order of their distance from the camera.) Unfortunately, this approach also limits parallelism by introducing dependencies: the background must be colored before the diagonals, which must be colored before the rectilinear lines. Since the students had been working on more complicated flags as part of the flag coloring programming assignment, they readily identified this issue and the difference between parallelizing the flag coloring for Mauritius and Great Britain.

To formalize this idea, the students were given the definition of a dependency graph (vertices are tasks and directed edges denote dependencies) and shown the example for coloring the flag of Great Britain shown in Figure 3.2. Then, to show their understanding, the students are asked to draw one for coloring the flag of Jordan (Figure 3.3).

3.3.1.5 Instructional Material

Materials for all these activities are in the repo corresponding to the section on the Unplugged Flag Maker Activity. Despite the name, this includes the plugged assignment materials, instructions and slides for the unplugged activity, and then the additional slides

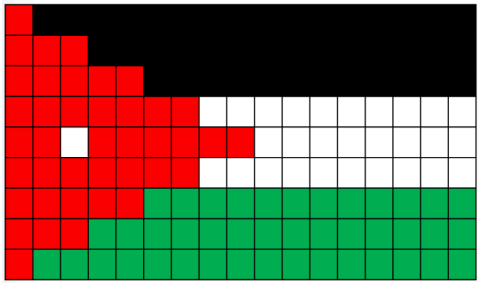


Figure 3.3: Flag coloring assignment version of the flag of Jordan

after the unplugged activity.

Another resource that might be useful is a video explaining data parallelism using parallel pixel coloring. It's explained as the difference between graphics with a CPU (one pixel per cycle) and a GPU (all pixels in a cycle). This is suggested as a possible pre-class viewing for students.

3.3.1.6 Experience and Teaching Tips

- Change some of the flags each term. It is possible to reuse flags between terms, but do not use the exact same assignment, or else students will acquire the assignment from the previous term and submit it. Thus far we have not observed students getting flags from previous terms, figuring which flags were reused, and copy and pasting just those small snippets of code.
- Demonstrate how to solve problems in different ways. For example, the Lithuanian flag shown in Figure 1.10 can be implemented with one set of nested loops with 3 set color statements, or three sets of nested loops, each with one set of set color statements. Be sure students see both ways to solve this problem.

3.3.1.7 Data and Analysis

In Fall 2024, we first used this activity in a CS1 class of 65 students split into three sections. To evaluate the activity, we collected the dependency graphs that students drew for coloring the flag of Jordan (see Figure 3.3). These were collected anonymously; the students were told that submission was voluntary and would have no impact on their grade. A total of 29 drawings (45% response rate). The first section is underrepresented in the count (only 4 drawings submitted), likely because the other activities ran long and they had less time to draw.

We examined the submissions to evaluate the level of understanding demonstrated. Our intended solution for the problem is shown in Figure 3.4; the stripes form the first layer and must be drawn first, followed by the red triangle, and then the white dot (a star in the actual flag). We accepted solutions that omitted the box for drawing the white stripe; in the Plugged-In Flag Maker Activity, the background is initially white so students had been "creating" white features by leaving that area blank.

The students who were at least mostly correct made up 59% of the respondents. Of the submissions, 10 (34%) were perfectly correct. Seven (24%) more were mostly correct. Five of

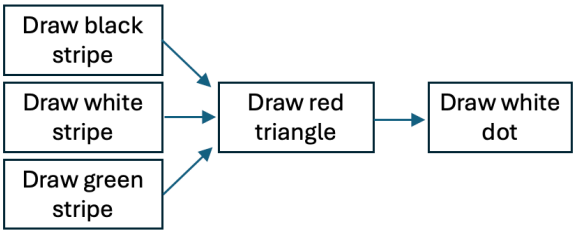


Figure 3.4: Dependency graph for coloring the flag of Jordan

these split the red triangle into upper and lower parts, which was again consistent with how some had been drawing such features in Plugged-In Flag Maker Activity, but which requires a more complicated dependency graph since the two halves have different dependencies. One of the other two mostly-correct submissions used one task for all the stripes and the other suggested the dependencies spatially but omitted the arrows.

Of the students who were not at least mostly correct, the most common error was to draw a linear chain of tasks. This suggests that they either thought about the graph in terms of sequential code or misunderstood the meaning of a dependency. Other submissions were incomplete but seemed to be working toward a linear solution and four (14%) students did not demonstrate any understanding; they drew the flag or started giving code to draw it.

Although we are reasonably happy with the level of understanding demonstrated, we plan to either spend more time on dependencies or include them in other assignments to reinforce this concept.

3.3.2 Knoxcraft

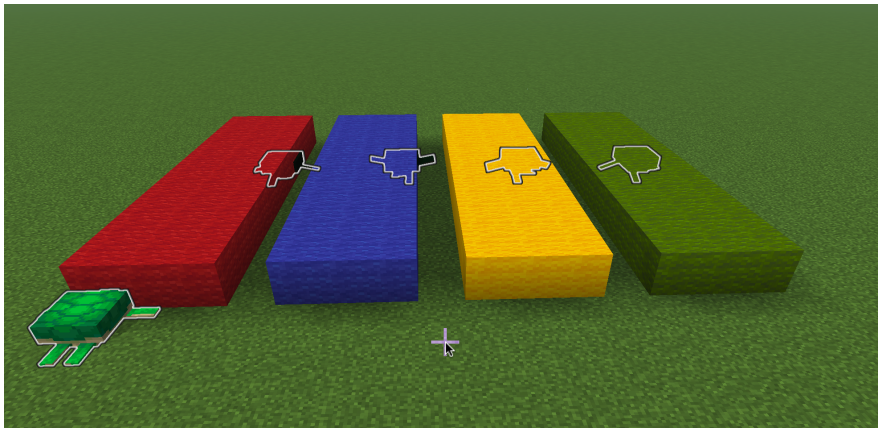
3.3.2.1 Activity Summary

This activity is inspired by Papert's Logo [44], where a 2D turtle walks around a 2D plane drawing pictures. Knoxcraft uses the voxel based world of Minecraft as a 3D canvas for a 3D version of Logo. Students write code in their IDE (VS Code, NetBeans, etc), call a special method to upload to the server, and then log into a specially modded Minecraft server to observe a turtle (literally a Minecraft turtle the Knoxcraft mod hijacks for this purpose).

3.3.2.2 Prerequisites

Knoxcraft requires the ability to call constructors and methods. Interesting creations will require loops and possibly conditionals. Recursion is a bonus, allowing the creation of recursive structures. The assignment involves connecting to a remote server, but the code for this is provided so students don't need to understand those calls. Students write all of their code in their IDE, and call an upload functions to upload to the server. The server doesn't actually run their code; it runs client side and produces a series of instructions in JSON for the server-side turtles to follow.

Figure 3.5: Parallel Knoxcraft Turtles.



3.3.2.3 Learning Outcomes

Knoxcraft enables parallelism. Students can create multiple "threads" of execution, which will map to different physical turtles. Figure 3.5 shows multiple Knoxcraft turtles building the flag of Mauritius in parallel. Multiple Knoxcraft "threads" build structures observably faster, giving students a visual demonstration of the value of parallelism.

3.3.2.4 Implementation Details

There are two main challenges for Knoxcraft:

- Faculty need to run a specially modded Minecraft server. This is not difficult, but it does require a bit of time and effort.
- Students need access to Minecraft accounts. Students can run programs uploaded by other students, so not every student needs a Minecraft account.

This assignment requires a larger time investment by faculty than the Greenfoot and Flag Maker assignments. We are working on a version with lower overhead for adoption.

3.3.2.5 Instructional Material

Materials for all these activities are in the repo corresponding to this section. This includes the background material and lab assignments.

3.3.2.6 Experience and Teaching Tips

- Give an example 3D structure, and require at least one 3D structure. One of our worked examples is a 2D flag, similar to what they have already done with the Flag Maker assignment, and unless we said otherwise, many students simply swapped in a different flag, skipping the 3D aspects of the project.
- Give a large structure with no threads, and a large structure with threads. Run both and have students time them.
- Require threads if you want students to use them.

- After giving this assignment, collect and share a "gallery" of images of the most interesting creations.

3.3.2.7 Data and Analysis

We did not collect data specifically on this assignment.

3.3.3 Greenfoot

3.3.3.1 Activity Summary

Greenfoot is described in detail in Section 1.4.2. It is a fun and intuitive 2D programming environment for making games or simulations that is accessible to novice Java programmers. It is made as an educational IDE for teaching Java, though we are using it in a course that primarily uses another environment.

3.3.3.2 Prerequisites

Greenfoot uses Java, though faculty teaching C++ at Casper College successfully taught their students to use Greenfoot for one or two assignments. Students need to understand methods, instance variables, classes, and loops. Knowledge of inheritance is helpful, but the use of subclasses is simple enough that CS1 students can use Greenfoot without understanding subclasses.

3.3.3.3 Learning Outcomes

Callbacks and event-driven programming. Greenfoot provides a mechanism to explicitly order the callbacks; otherwise the ordering is undetermined and can lead to unpredictable results.

3.3.3.4 Implementation Details

Greenfoot is available for Mac, Windows, or Linux, and is freely available for download and use.

3.3.3.5 Instructional Material

Materials for all these activities are in the repo corresponding to this activity. This includes the background material and both lab assignments, plus videos and "worked example" sample code to share with the students about creating games in Greenfoot.

3.3.3.6 Experience and Teaching Tips

- Give students open-ended assignments with a minimum set of requirements, but no maximum. Trust in the creativity of students to take ownership of their projects.
- Share "worked examples" with the students, such as the four sample projects provided in the project repo.

- Create a "gallery" of the most interesting submissions to share with the class, or to show the next year's class. Set high expectations and motivate students with their desire to perform well in front of their peers, not pressure to achieve grades.
- Disallow games with many examples already shared and uploaded to http://greenfoot.org, such as the snake game and flappy bird.
- Ask students to make a short video explaining and demonstrating the best features of their game. This helps build presentation skills, and videos can be shown to future classes.

3.3.3.7 Data and Analysis

We did not collect data specifically on this assignment.

3.4 Course-wide Pre and Post Course Surveys

In the Winter 2024 and Winter 2026 terms, we conducted pre and post surveys in our introductory course to assess student awareness and familiarity with programming concepts as well as parallel and distributed concepts. Our introductory course is designed to be taken by students with no background in computer science, meaning the responses are coming from pure novices. We conducted the pre-survey in the first week of the term, and the post-survey in the last week. A total of 19 students took both the pre and post-surveys in Winter 2024, while 25 students took both surveys in Winter 2026. Questions were scored on a 5-point Likert scale ("No experience at all", "Slightly Experienced", "Moderately Experienced", "Very Experienced" and "Extremely Experienced"). We test each item using the Wilcoxon signed-rank test on matched pairs and report effect sizes as Cliff's delta. Thus, p-values are exploratory/unadjusted. Our results for Winter 2024 are presented in Table 3.1 and those for Winter 2026 are in Table 3.2.

Table 3.1: CS1 Pre/Post Survey Analysis Results (Winter 2024)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Programming Concepts									
Data/Variable Types	19	3.16	4.37	1.21	0	0.001	Yes	0.58	large
Arithmetic Expr. and Operators	19	3.42	4.21	0.79	6	0.008	Yes	0.40	medium
Branching	19	3.32	4.16	0.84	4	0.004	Yes	0.44	medium
Loops	19	3.26	4.11	0.84	4	0.004	Yes	0.44	medium
Calling functions/methods	19	3.16	4.11	0.95	3	0.003	Yes	0.48	large
Arrays	19	2.58	3.95	1.37	0	0.001	Yes	0.63	large
Writing static functions	19	2.53	3.95	1.42	0	0.001	Yes	0.66	large
Writing methods	19	2.89	4.21	1.32	0	0.001	Yes	0.62	large
Writing classes	19	2.58	4.05	1.47	0	0.001	Yes	0.67	large
Creating objects	19	2.68	3.89	1.21	0	0.001	Yes	0.58	large
PDC Concepts									
parallel	19	1.84	2.89	1.05	3	0.002	Yes	0.58	large
distributed	19	1.74	2.47	0.74	6	0.009	Yes	0.43	medium
event	19	2.37	2.84	0.47	17.5	0.071	No	0.28	small

Table 3.2: CS1 Pre/Post Survey Analysis Results (Winter 2026)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Programming Concepts									
Data/Variable Types	25	4.96	5.80	0.84	16.5	0.002	Yes	0.44	medium
Arithmetic Expr. and Operators	25	5.12	5.80	0.68	21.0	0.008	Yes	0.34	medium
Branching	25	5.20	5.80	0.60	24.0	0.015	Yes	0.31	small
Loops	25	4.76	5.72	0.96	14.0	0.001	Yes	0.45	medium
Calling functions/methods	25	4.88	5.76	0.88	12.0	0.001	Yes	0.47	medium
Arrays	25	3.44	5.32	1.88	0.0	0.000	Yes	0.70	large
Writing static functions	25	3.56	5.52	1.96	0.0	0.000	Yes	0.74	large
Writing methods	25	3.80	5.52	1.72	0.0	0.000	Yes	0.66	large
Writing classes	25	3.56	5.44	1.88	0.0	0.000	Yes	0.70	large
Creating objects	25	3.72	5.52	1.80	0.0	0.000	Yes	0.68	large
PDC Concepts									
parallel	25	2.92	4.16	1.24	26.5	0.002	Yes	0.49	large
distributed	25	2.72	3.76	1.04	31.0	0.005	Yes	0.42	medium
event	25	3.32	4.08	0.76	48.0	0.028	Yes	0.30	small

We see significant effects for all of the programming concepts in Winter 2024 and all but one in Winter 2026, which would be expected in an introductory course aimed at novices with little to no background with any of these concepts. Another way to think about this: If students did not report significant improvements in their experiences with the core programming concepts presented in these results, we would need to re-evaluate the entire course. Similarly, we see significant effects for parallel and distributed computing, but not for event-driven programming. This makes some intuitive sense, as the event-driven programming module, which relies heavily on Greenfoot, may seem event-driven to an expert who understands how event-driven loops work in GUI code, while a novice may not have the vocabulary or understanding to see the connections between events and Greenfoot.

We also asked students at the end of the Winter 2024 term to rate how well a series of resources and activities helped them learn in CS1, using a 5-point Likert scale. The results are presented in Table 3.3. Students find the most interactive aspects of the class (lab periods, the ZyBooks interactive textbook, and in-class activities) to be the most useful to their learning, which fits with the "active learning" approach we have adopted across our curriculum.

Table 3.3: Post-Survey Course Resource and Activity Helpfulness Ratings (1–5 Scale)

Resource / Activity	n	Mean	Std	Median
Course Resources & Activities				
Lecture	19	4.00	0.75	4.0
Lab	19	4.32	0.75	4.0
In Class Activities	19	4.05	0.71	4.0
Class Attendance	19	3.68	0.95	4.0
Lab Attendance	19	3.89	1.05	4.0
Office Hours with Instructor	19	3.84	1.01	4.0
Office Hours with TAs	19	3.68	1.25	4.0
College Tutoring Services	19	3.84	1.12	4.0
Preparing for Exams	19	3.89	0.94	4.0
ZyBooks	19	4.05	0.71	4.0

Finally, for the Winter 2026 pre- and post-surveys, we asked students to report their knowledge and comfort level with various topics, including PDC topics. We present the results in Table 3.4. We observe moderately significant improvements in general understanding of computing and PDC, along with no significant changes to student interest in computing, interest in PDC, or the importance of PDC. This suggests that the new changes to the curriculum do not decrease student interest in computing in any significant way, while improving student self-perception of their understanding of both computing and PDC. These results are promising.

Table 3.4: CS1 Pre/Post Computing Attitudes Analysis Results (Winter 2026)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Computing Attitudes									
computing interest	25	5.44	5.16	-0.28	60.5	0.443	No	−0.15	small
PDC interest	25	5.08	4.88	-0.20	62.0	0.582	No	−0.10	negligible
PDC importance	25	5.28	5.20	-0.08	49.5	0.778	No	−0.04	negligible
computing understanding	25	4.76	5.56	0.80	16.5	0.007	Yes	0.41	medium
PDC understanding	25	4.04	4.96	0.92	18.0	0.005	Yes	0.46	medium

3.5 Conclusion

We are very happy with our new PDC-infused CS1 course sequence. A major concern with changing CS1 is that the introductory course is so full that any new content pushes out existing content. We did replace some material we had taught for many years, but found that this was less of an issue than we expected, in part because the concepts taught in that material were adequately covered in other assignments.

Changing CS1 at many institutions is challenging for two main reasons: The first is that the course is packed with material and the perception is that adding anything new requires removing something else. The second reason is that CS1 has a large number of stakeholders; every subsequent course in the department depends on what students learn in CS1, not to mention courses in other departments that may also use CS1.

To overcome the first of these issues, we reflected on the assignments and realized that we had some duplicated coverage (of nested loops). By reducing this, we freed up time for more exciting assignments. For the second issue, we had a structural advantage. At Knox College, we are a small department (4 faculty, 2 of whom are on this project), so we can change CS1 without as much discussion. Having small classes makes it easier for us to fill in any student knowledge gaps that appear in subsequent courses. We are also highly motivated because we feel like teaching a modern CS1 course is important enough to take risks. This is the fundamental truth: the realization that the status quo is unacceptable should motivate change. Even in a larger department where it's harder to build consensus around changes, build from that and start, even if it's just a single topic or assignment.

3.5.1 Experiences and Advice

Our advice for faculty incorporating PDC topics into CS1 is as follows:

- **Think about existing assignments in new ways.** We have taught some version of **Flagmaker** for over 15 years, but only realized recently that the replay thumb (as shown in Figure 1.11) can demonstrate parallelism.
- **Great assignments with good topic coverage are better than good assignments with great topic coverage.** We added Greenfoot to the end of the term to replace prior assignments on classes, and Java GUI programming that used event-driven programming and callbacks. Those older assignments covered those topics very precisely, but were not very fun or interesting. With our previous assignments, no student ever showed their roommate that they had implemented instance variables correctly; however, students often show their roommate their Greenfoot project, not because it uses instance variables correctly, but because they are proud or excited or passionate about what they have built. Using instance variables correctly is not the goal, but rather a mechanism to achieving a different goal. We have found that we had been over-thinking our assignments, and not trusting the students enough.
- **Manage the amount of configuration students navigate and manage.** We expected that running a modded Minecraft server for Knoxcraft would be the biggest challenge; instead, we found it far more challenging to help students install the proper version of Minecraft, log into younger siblings' XBox Live accounts, and so on. When added onto the already existing challenges of installing VS Code and Java, choosing the correct Flagmaker JavaFX library (easy but still something students can get wrong), using custom-written codingbat.com exercises (another service we use that requires basic configuration), installing Greenfoot, creating a Runestone Academy account for our interactive online textbook, and so on, the configuration burden adds up. We had created a client-server assignment where students would use Greenfoot to build a Connect 4 client that connects to a REST server; it's a great assignment but was too much configuration.
- **Embrace open-ended assignments.** This is probably the most counter-intuitive lesson we have learned from integrating PDC into our CS1 course. Our assignments used to be tightly specified and carefully written to eliminate ambiguity. While this helps teaching and building fundamentals, eliminating ambiguity also tends to create a ceiling on how much interesting work a student might do. Open ended assignments, such as building a game in Greenfoot, or creating a 3D structure in Knoxcraft, gives students meaningful opportunities to flex their creativity. In retrospect, we have realized how *few* opportunities for creativity we had in our course.
- **Change things even if a new assignment might fail.** Before we landed on Greenfoot and Knoxcraft, we asked students to use LLMs to create fifteen events in a JSON format, upload them to a shared event server, download the full library of events, and then use basic data science code to compute aggregate statistics. How many people created holidays? How many copies of the Battle of Hastings in 1066 were uploaded? What was the most popular year for events? This sounded like a brilliant idea when we created the assignment.
  It is hard to overstate how thoroughly *uninterested* students were in creating events or analyzing patterns in the events that were uploaded by other students. We had to admit that this assignment, while noble in its lofty goal of introducing a distributed shared

server to students, was not actually fun or interesting and needed to be replaced. This failure forced us to experiment with other ideas, and eventually led us to Knoxcraft (a more interesting shared server) and adding additional content with Greenfoot.

Appendix

GitHub Link to Instructional Material

Materials for all the activities in this chapter are in our GitHub repository.

Detailed Pre and Post Syllabus

Also included are a course syllabus and a day-by-day summary of what was taught (with links to the materials used and the changes highlighted in **bold**).

Organization of Material

The GitHub repository contains the full course with the modifications described in this chapter (as taught in Fall 2025). `COURSE_CALENDAR.md` depicts the day by day presentation of material and due dates for assignments. This document has links to the actual materials, which are stored in the `Archive` directory. The directory `Detailed_Pre_and_Post_Syllabus` contains the course syllabus.

The materials for the three activities described in this chapter are in

- `Unplugged_Activities/flagmaker` (despite the name, this directory contains both the plugged and unplugged versions of Flagmaker),
- `Plugged_Activities/knoxcraft` and
- `Plugged_Activities/greenfoot`.

Survey and Other Anonymized Data and Analysis

Student-level survey data are not publicly released, but aggregate results are reported in the chapter.

Chapter 4

CS1 Course Package – University of Southern Indiana

Structured Java Activity Adoption with Evidence-Rich Unplugged and Plugged-In Modules[†]

Srishti Srivastava

University of Southern Indiana

fsrishti@usi.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 126
4.1 Introduction . . . 130
4.2 How CS1 was changed and PDC topics integrated . . . 131
4.2.1 Integrated Syllabi (Pre and Post) . . . 131
4.2.2 Highlights . . . 131
4.2.3 Challenges . . . 133
4.3 New Unplugged Activities . . . 134
4.3.1 Unplugged Flag Maker Activity . . . 134
4.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 134
4.3.1.2 Implementation Details . . . 134
4.3.1.3 Experience and Teaching Tips . . . 136
4.3.1.4 Data and Analysis . . . 136
4.3.2 Unplugged Penny Sorting Exercise . . . 138
4.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 138
4.3.2.2 Implementation Details . . . 138
4.3.2.3 Experience and Teaching Tips . . . 139
4.3.2.4 Data and Analysis . . . 140
4.4 New Plugged-in Activities . . . 141
4.4.1 Flag Maker Plugged-In Activity . . . 141
4.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 141
4.4.1.2 Implementation Details . . . 143
4.4.1.3 Experience and Teaching Tips . . . 144
4.4.1.4 Data and Analysis . . . 144
4.4.2 Parallel Penny Sort Plugged-In Activity . . . 144
4.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 144
4.4.2.2 Implementation Details . . . 145
4.4.2.3 Experience and Teaching Tips . . . 146
4.4.2.4 Data and Analysis . . . 147
4.4.3 Greenfoot Plugged-In Demo . . . 147
4.4.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 147
4.4.3.2 Implementation Details . . . 148
4.4.3.3 Experience and Teaching Tips . . . 149
4.4.3.4 Data and Analysis . . . 149
4.5 Course-wide Pre and Post Course Surveys . . . 149
4.5.1 Results and Analysis: Fall 2024 Benchmark Survey Data . . . 150
4.5.2 Results and Analysis: Fall 2025 PDC Intervention Survey Data . . . 152
4.5.3 Did the PDC Intervention Improve Learning? . . . 154
4.6 Other Activities and Ideas . . . 155
4.6.1 More Processors are Not Always the Best . . . 155
4.7 Conclusion . . . 158
4.7.1 Future Work . . . 159
Appendix . . . 160

Chapter figures

4.1 USI CS1 Pre vs Post Percentage of Correct Questions for N=24 . . . 137
4.2 USI CS1 Response Distribution by Item . . . 138

4.3 USI CS1 Thematic Analysis of Open Response Survey Questions . . . 139
4.4 USI CS1 Diverging Likert Distribution of Engagement Items (n = 33) . . . 141
4.5 USI CS1 Example Flag Represented as a Colored Grid and Corresponding Code . . 142
4.6 USI CS1 Fall 2024 Pre to Post Shift on Every Survey Item . . . 151
4.7 USI CS1 Fall 2025 Pre to Post Shift on Every Survey Item . . . 153
4.8 USI CS1 Improvement of Each Item 2024-25 . . . 155
4.9 USI CS1 Pre to Post Gains . . . 156
4.10 USI CS1 Progressive Operational Activity Phases . . . 158

Chapter tables

4.1 USI CS1 Relevant PDC Topics and Blooms Levels . . . 127
4.2 USI CS1 Gender Distribution . . . 129
4.3 USI CS1 Race/Ethnicity Distribution . . . 129
4.4 USI CS1 Class Standing . . . 129
4.5 USI CS1 Declared Majors . . . 129
4.6 USI CS1 Self-Reported General Computing Experience . . . 129
4.7 USI CS1 Self-reported prior programming experience of CS1 participants . . . 130
4.8 USI CS1 Fall 2025 Course Syllabus Calendar with Highlighted PDC Content . . . 132
4.9 USI CS1 Approximate Timing of Flag Maker Activity . . . 135
4.10 USI CS1 Approximate Timing of Unplugged Penny Sorting Exercise . . . 140
4.11 USI CS1 Approximate Timing of the Flag Maker Plugged-In Activity . . . 144
4.12 USI CS1 Approximate Timing of the Penny Sort Plugged-In Activity . . . 146
4.13 USI CS1 Approximate Timing of the Greenfoot Plugged-In Demo . . . 148
4.14 USI CS1 Pre/Post Survey Results from Benchmark Fall 2024 . . . 150
4.15 USI CS1 Pre/Post Survey Results from PDC Intervention Fall 2025 . . . 152
4.16 USI CS1 Combined Pre/Post Survey Results . . . 157
4.17 USI CS1 Approximate Timing of the Unplugged “More Processors Are Not Always the Best” Activity. . . . 157

Abstract

The growing importance of parallel and distributed computing (PDC) in modern computing systems calls for early exposure to PDC concepts in undergraduate computer science education. Yet, introductory programming courses such as CS1 have traditionally focused on sequential programming paradigms, leaving students under prepared for the realities of contemporary computing environments. This chapter presents a course exemplar developed at the University of Southern Indiana (USI), in which a CS1 course titled, "Introduction to Object-Oriented Programming", was redesigned to include PDC concepts throughout a semester via experiential and active learning pedagogy. This chapter describes the motivations, design decisions, and structural changes that guided the course redesign, detailing how PDC principles were incrementally and meaningfully integrated alongside core object-oriented programming concepts. Through carefully crafted hands-on activities, collaborative exercises, and project-based learning experiences, students are introduced to foundational PDC ideas in a contextualized and accessible manner from the very beginning of their computing education. The chapter further shares insights drawn from the implementation of this redesigned course, discussing observed impacts on student engagement, conceptual understanding, and programming competency. Practical recommendations are offered for instructors seeking to undertake similar curriculum transformations in their own CS1 courses, with the aim of equipping the next generation of programmers with the parallel and distributed computing mindset that modern software development demands.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Starting Out With Java: Control Structures through Objects, 8th edition, Pearson Revel 2022, by Tony Gaddis.

Relevant core courses: CS258 (CS1): Introduction to Object Oriented Programming

Pre-requisites: At USI, students enrolling in CS1 are required to have completed or be concurrently enrolled in linear algebra as a prerequisite or co-requisite. In practice, the majority of incoming students satisfy this requirement through placement scores prior to beginning the program.

Relevant PDC topics: Table 4.1 lists the integrated PDC topics with bloom's classification (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV), and Creating (CR)).

Learning outcomes from PDC interventions:

1. FlagMaker Activity : Data Parallelism (Plugged-In& Unplugged Activity)
   (a) Define the term data parallelism as it applies to distributing a computational workload across multiple processors.
   (b) Distinguish between sequential computing and parallel computing using examples from the flag-drawing task.

Table 4.1: USI CS1 Relevant PDC Topics and Blooms Levels

PDC Concept	Chapter Section				
	Unplugged Flag Maker Activity (4.3.1)	Unplugged Penny Activity (4.3.2)	Plugged-In Flag Maker Activity (4.4.1)	Plugged-In Parallel Sort Penny Activity (4.4.2)	Greenfoot Activity (4.4.3)
Data/Task Parallelism	RE, UN, AN	RE, UN, AN	RE	RE, UN, AP, AN, EV	RE
Shared Memory	RE, UN	RE, UN	RE	RE, UN	RE
Scheduling/Load Balancing	RE, UN, AN	RE, UN, AN	RE	RE	RE
Speedup/Amdahl's Law	RE, UN, AN	RE, UN, AN	RE	RE, UN, AN, EV	RE
Event Handling					RE, UN

(c) Explain why parallel execution time is bounded by the slowest processor rather than the average processor time.
(d) Recall the definitions of key PDC vocabulary including multicore, multiprocessor, synchronization, and scheduler.
(e) Explain what contention means when multiple processors compete for a shared resource.
(f) Compare the completion times recorded across the single-processor, two-processor, and four-processor scenarios of the unplugged activity.
(g) Apply loop-based indexing to correctly partition a flag grid into horizontal stripes of specified heights.
(h) Use conditional logic within a loop to assign the correct color to each cell based on its row position.
(i) Differentiate between a row-based data partition and a column-based data partition in terms of data access patterns.

2. Penny Sorting Activity : Data/Task Parallelism, Speedup, and Amdahl's Law (Unplugged & Plugged-In Activity)
   (a) Define speedup as the ratio of sequential execution time to parallel execution time for the same task.
   (b) Describe what load imbalance means when one processor receives significantly more work than others.
   (c) Explain scalability as a parallel system's ability to increase performance proportionally as more processors are added.
   (d) Identify which phase of the penny sorting activity models a load-imbalanced parallel execution.
   (e) Compute speedup values from timing data collected during the sequential and parallel phases of the penny sorting activity.
   (f) Implement a sequential sort in Java using a selection sort, insertion sort, or bubble sort algorithm to order an array of integer penny years.
   (g) Use *Arrays.parallelSort* in Java to perform a parallel sort on the same array of penny years.
   (h) Differentiate between data parallelism and task parallelism by identifying

which model applies to each phase of the penny sorting activity.

(i) Examine how the unequal distribution of pennies in Phase 3 of the unplugged activity increases overall parallel execution time despite involving more processors.

(j) Contrast the theoretical speedup predicted by Amdahl's Law with the observed speedup from the recorded timing data.

(k) Explain why adding more processors does not always result in proportionally faster execution.

3. Greenfoot Plugged-In Activity : A Sneak Peak into Event Handling

   (a) Identify event-driven programming as a paradigm in which program execution is determined by user-generated or system-generated events.

   (b) Describe how an event listener responds to a specific event such as a key press or mouse click in a Greenfoot scenario.

   (c) Recognize the role of actors and the world class in structuring a Greenfoot program.

Context for use: The PDC modules described in this chapter were implemented in CS 258, Introduction to Object-Oriented Programming, a CS1 course at the University of Southern Indiana (USI), a public master's-level institution located in Evansville, Indiana. USI enrolls approximately 9,750 students across more than 130 areas of study and holds Carnegie classifications as a community-engaged institution. The Computer Science program resides within the College of Business and comprises seven full-time faculty members. CS 258 is a mandatory CS1 course required of all Computer Science and Computer Information Science majors as well as CS minors. The programming language used for the course is Java. The course introduces the fundamental concepts of programming from an object-oriented perspective, covering data abstraction, inheritance, overriding, programming flow of control, operator precedence, and introductory data structures such as lists and arrays. It also addresses human-computer interfaces, graphics, and the social implications of computing. In a semester, the course is taught over 16 weeks with 50-minute or 75-minute sessions held thrice or twice per week. The class enrolls approximately 35–40 students on average. Students are primarily at the freshman level, with prior programming experience ranging from none to introductory. Table 4.4 shows the classification of students taking the CS1 course across Fall 2024 (baseline semester) and Fall 2025 (PDC intervention semester). Table 4.7 shows incoming programming experience of students entering CS1. The class included a mix of Computer Science majors, Computer Information Science majors, and Mathematics majors completing a double major with Computer Science. The student distribution by majors is shown in Table 4.5. Almost eighty percent of the students in CS1 identify as male and approximately 13.4% identify as female as shown in Table 4.2 and student race is shown in Table 4.3. Table 4.6 shows incoming student experience of general computing such as internet searches, email, discord, discussion groups, games, word processing, spreadsheets, organizing files, and organizing folders. With respect to prior knowledge of PDC, no baseline PDC instruction had been provided to students before the modules described here.

Table 4.2: USI CS1 Gender Distribution

Gender	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
Male	24	29	53 (79.1%)
Female	2	7	9 (13.4%)
Other	0	1	1 (1.5%)
No Response	0	4	4 (6.0%)

Table 4.3: USI CS1 Race/Ethnicity Distribution

Race/Ethnicity	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
White / Caucasian	20	30	50 (74.6%)
Black / African American	3	4	7 (10.4%)
Hispanic	1	3	4 (6.0%)
Asian / Pacific Islander	2	1	3 (4.5%)
Two or more ethnicities	0	1	1 (1.5%)
No Response	0	2	2 (3.0%)

Table 4.4: USI CS1 Class Standing

Class Standing	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
First year (freshman)	11	14	25 (37.3%)
Sophomore	8	13	21 (31.3%)
Junior	7	13	20 (29.9%)
Senior	0	1	1 (1.5%)

Table 4.5: USI CS1 Declared Majors

Major	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
Computer Science	22	40	62 (92.5%)
Computer Information Systems	3	1	4 (6.0%)
Computer Engineering	0	1	1 (1.5%)
Math	1	0	1 (1.5%)
Other (biology)	1	0	1 (1.5%)

Students could select multiple majors; percentages are of students and therefore sum to more than 100%.

Table 4.6: USI CS1 Self-Reported General Computing Experience

Computing Experience	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
No experience at all	1	0	1 (1.5%)
Very little experience	1	0	1 (1.5%)
Slightly experienced	0	3	3 (4.5%)
Between slightly and moderately experienced	3	5	8 (11.9%)
Moderately experienced	5	6	11 (16.4%)
Very experienced	12	11	23 (34.3%)
Extremely experienced	4	16	20 (29.9%)

Table 4.7: USI CS1 Self-reported prior programming experience of CS1 participants

Programming Experience	Fall 2024 (n=26)	Fall 2025 (n=41)	Total (n=67)
No experience at all	2	4	6 (9.0%)
Very little experience	4	11	15 (22.4%)
Slightly experienced	7	11	18 (26.9%)
Between slightly and moderately experienced	2	5	7 (10.4%)
Moderately experienced	9	7	16 (23.9%)
Very experienced	1	3	4 (6.0%)
Extremely experienced	1	0	1 (1.5%)

4.1 Introduction

Many instructors teaching CS1 courses face a genuine and understandable challenge that the course already carries a substantial conceptual load, covering object-oriented design, control flow, data structures, and software engineering fundamentals, and the prospect of adding PDC content without displacing core learning objectives can feel daunting. As a result, PDC integration in CS1 continues to be the exception rather than the norm. This chapter is written for instructors who share that concern but are looking for practical, tested strategies to resolve it. Specifically, it is written for instructors at institutions similar to USI, a mid-sized public master's university with a community-engaged mission, where resources are not unlimited, class sizes are modest, and students arrive at CS1 with widely varying levels of prior programming experience. The strategies described here do not require additional instructional time beyond a standard 50-minute to 75-minute class session, do not presuppose access to parallel computing hardware or specialized software environments, and are designed to complement rather than compete with the existing learning objectives of an introductory object-oriented programming course.

The approach taken in this work is grounded in active learning and the use of unplugged activities as bridges to computational thinking. Unplugged activities, in which students physically enact computational processes without the use of a computer, have a well-established history in CS education as tools for making abstract concepts tangible and accessible. Their effectiveness is particularly well-documented at the introductory level, where students are simultaneously learning to program and to think algorithmically, and where the cognitive distance between a concept and its implementation can otherwise be prohibitively large [4, 32, 40, 54]. By first giving students a physical, embodied experience of parallel execution, and then connecting that experience to a programming task that operationalizes the same concepts, the modules described in this chapter follow a concrete-to-abstract pedagogical sequence that is also consistent with Bloom's taxonomy of cognitive development.

Three PDC modules containing five activities are presented. The first module focuses on data parallelism and uses a flag-drawing activity, first as a loop-based Java programming assignment implementing serial flag coloring, followed by an unplugged paper-and-marker form, to illustrate how a computational task can be partitioned across multiple processors. The second module introduces data and task parallelism, speedup, and Amdahl's Law through a penny sorting and searching activity, again bridging an unplugged experience to a plugged-in implementation using Java arrays and parallel sorting methods. The third module provides

a brief introduction to event-driven programming through a demonstration of Greenfoot, offering students a preview of paradigms they will encounter more formally in their follow-up CS2 course.

4.2 How CS1 was changed and PDC topics integrated

A central design principle of the approach described in this chapter is that PDC concepts can be integrated into a CS1 course without restructuring the syllabus, displacing core content, or requiring additional class sessions. Rather than treating PDC as a standalone topic that competes for time alongside the traditional CS1 curriculum, the integration strategy adopted here embeds PDC thinking directly into the instructional contexts where foundational programming concepts are already being taught. The result is a course that looks, from a structural standpoint, nearly identical to a conventional CS1 offering, but in which selected modules are taught through a lens that naturally introduces parallel reasoning.

4.2.1 Integrated Syllabi (Pre and Post)

The course calendar from the syllabus for CS1 (Fall 2025) at USI as shown in Table 4.8. It spans 16 weeks across nine content modules, covering the standard CS1 arc from introductory programming and Java fundamentals through decision structures, loops, methods, classes and objects, and arrays. The PDC interventions, highlighted in the course calendar, are threaded into this existing structure at three points, each chosen because the underlying CS1 content provides a natural and mutually reinforcing context for the PDC concept being introduced.

4.2.2 Highlights

The vast majority of the CS1 course structure remained unchanged. All nine content modules were retained in their original form and sequence, covering the same textbook chapters, the same auto-graded programming projects, and the same quiz schedule. The course continued to use the Pearson Revel platform for readings and textbook-aligned assessments, and no existing quiz, exam, or core programming assignment was removed from the calendar to make room for PDC content.

In Week 6, one of the standard Chapter 4 programming assignments was replaced by the Flag Maker Plugged-In Activity. This substitution was the only deletion made to the existing assignment structure. This activity was designed to assess the same loop-based programming skills that the displaced assignment would have evaluated, so the change carried no loss in coverage of the Chapter 4 learning objectives. It did, however, add a new in-class component in the form of the Unplugged Flag Maker Activity, which was conducted during a regularly-scheduled class session in Week 6 without extending beyond the allotted 75 minutes.

In Weeks 15 and 16, the course's existing final project slot was repurposed. Rather than assigning a generic open-ended or object-oriented capstone project as might be conventional in a CS1 course, the Parallel Penny Sort Plugged-In Activity was designated as the final

Table 4.8: USI CS1 Fall 2025 Course Syllabus Calendar with Highlighted PDC Content

Modules	Week	Assigned Reading	Assignments	Quizzes and Exams
Module 1: Introduction to Computers and Java	Week 1	Syllabus, Start Here Content, Instructor Notes, Textbook Chapter 1 (Pearson Revel)	Chapter 1 programming challenge and Pre-Course Experience Survey	Textbook Quiz 1.3, 1.4, 1.5, 1.6
Module 2: Java Fundamentals (Variables, Data Types, Arithmetic Operators, Comments, Console input output, etc.)	Week 2	Textbook Chapter 2 (Pearson Revel)	Chapter 2 Programming Project 1 and 2	Textbook Quiz 2.1, 2.2, 2.3, 2.4, 2.5, 2.7, 2.8, 2.9, 2.10, 2.13, 2.15
Module 3: Decision Structures	Weeks 3-4	Textbook Chapter 3 (Pearson Revel)	Chapter 3 Programming Project 1 and 2	Textbook Quiz 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.8, 3.9
Module 4: Loops and Files	Weeks 5-6	Textbook Chapter 4 (Pearson Revel)	Chapter 4 Programming Project 1, 2 and 3 AND Unplugged Flag Maker Activity (4.3.1) and Flag Maker Plugged-In Activity (4.4.1) as midterm assignment)	Textbook Quiz 4.1, 4.2, 4.4, 4.5, 4.6, 4.7, 4.9, 4.10
Module 5: Methods	Weeks 7-8	Textbook Chapter 5 (Pearson Revel)	Chapter 5 Programming Project 1 and 2	Textbook Quiz 5.1, 5.2, 5.4
Module 6: A First Look at Classes	Weeks 9-10	Textbook Chapter 6 (Pearson Revel)	Chapter 6 Programming Project 1 and 2	Textbook Quiz 6.1, 6.2, 6.3, 6.4
Module 7: Arrays and ArrayList Class	Weeks 11-12	Textbook Chapter 7 (Pearson Revel)	Chapter 7 Programming Project 1 and 2	Textbook Quiz 7.1, 7.3, 7.4 , 7.8, 7.9, 7.10, 7.11, 7.13
Module 8: A Second Look at Classes and Objects	Weeks 13-14	Textbook Chapter 8 (Pearson Revel)	Chapter 8 Programming Project 1, 2, and 3	Textbook Quiz 8.1, 8.3, 8.4, 8.5
Module 9: Final Project Work Weeks/ Lab Days	Weeks 15-16	Greenfoot Plugged-In Demo (4.4.3) (Event Driven programming)	Unplugged Penny Sorting Exercise (4.3.2) and Parallel Penny Sort Plugged-In Activity (4.4.2) Due as Final Project	Post-Course Experience Survey

project deliverable. The project submission deadline and the final exam day remained unchanged. The Greenfoot Plugged-In Demo, delivered in the same period, replaced what would otherwise have been an unstructured lab or review session during the final project work weeks, requiring no displacement of any assessed content. The pre- and post-course experience surveys, administered in Weeks 1 and 16 respectively, were added as brief in-class activities and required approximately ten minutes of class time each, well within the buffer that naturally exists at the start and end of a semester.

4.2.3 Challenges

The challenges encountered during PDC intervention are described as: challenges of adoption, challenges of adaptation, and challenges arising from the research layered on top of instruction.

Adoption challenges: the most immediate obstacle was the in-class coordination of the unplugged activities. Both the Unplugged Flag Maker Activity and Unplugged Penny Sorting Exercise require physical materials, group formation, and timed phases that must be orchestrated within a fixed class period. In our sections of 28–34 students meeting in a standard lecture room, distributing materials, forming groups of five to six, and running three timed phases of Penny Sorting while reserving time for a substantive post activity discussion required careful minute by minute planning in advance. Instructors adopting these activities should budget preparation time accordingly and should not assume that transitions between phases will absorb themselves into the period.

Adaptation challenges: The activities described here are calibrated to a particular room, roster size, and staffing level. Larger sections lengthen the group formation step disproportionately, and rooms with fixed furniture may make the implementation more time-consuming than anticipated. Staffing is the other adjustable variable: having a teaching assistant present to distribute penny bags, manage timing, and circulate between groups during Phases 2 and 3 of Penny Sorting is strongly advisable, and we regard it as close to necessary in sections larger than 30 students. Instructors working under constraints unlike ours should expect to modify phase timings rather than the phase structure itself.

Research challenges: Administering the pre- and post-activity assessments in a way that preserved their research validity without disrupting the flow of the session proved to be a distinct challenge from teaching the activities themselves. The Flag Maker pre-test had to be completed before any PDC content was introduced, which constrained the sequencing of the class period and required explaining to students why they were being asked about material they had not yet encountered. Some students found this ambiguity uncomfortable, and a few initially misread the pre-test as a graded quiz. Instructors should state explicitly, both verbally and in writing, that pre-test responses carry no grade consequence and that incorrect answers are expected and informative. Moreover, instructors intending to collect data for publication should leave sufficient buffer time before the semester begins to secure IRB approval at their own institutions.

4.3 New Unplugged Activities

4.3.1 Unplugged Flag Maker Activity

4.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

The Unplugged Flag Maker Activity introduces students to PDC concepts through collaborative paper-based simulations. By progressively moving from sequential execution to parallel processing and resource contention scenarios, the activity helps students build intuition about speedup, scalability, load balancing, and synchronization while promoting active learning and student engagement. The activity is designed for CS1 level students and requires minimal technical prerequisites. Students should have:

- Basic familiarity with how programs execute sequentially.
- An introductory understanding of processes and computing systems.

The **activity materials** are intentionally simple and include:

- Gridded paper to serve as flag sheets
- Markers or crayons (colors used in this activity: red, blue, yellow, and green)
- Student participants assigned roles as processors and a CPU scheduler (usually the instructor), and timer

As for the **learning outcomes**, by the end of the module, students should be able to:

1. Explain Von Neumann sequential execution and CPU scheduling.
2. Describe multi-core processing, concurrency, and data parallelism.
3. Understand and apply data decomposition and load balancing strategies.
4. Calculate and interpret speedup and discuss scalability.
5. Recognize the importance of communication and coordination among parallel processes.
6. Identify and analyze resource contention, file contention, and mutual exclusion.
7. Evaluate trade-offs between increased parallelism and system performance.

4.3.1.2 Implementation Details

The material used for implementing this activity can be found at this GitHub link. The activity consists of four sequential activities that simulate increasingly complex computing environments:

- Phase I: Sequential Execution: One student acts as a process and another as a scheduler. Students simulate single-processor execution by coloring all cells in a grid. Introduces sequential execution and CPU scheduling.
- Phase II: Dual-Processor Parallel Execution: Two students work in parallel while a scheduler coordinates tasks. Students divide the grid into sections, illustrating data decomposition and parallel execution. Introduces concurrency, load balancing, and speedup.
- Phase III: Four-Processor Parallel Execution: Four students execute tasks simultaneously. Students observe the benefits and limitations of scaling parallel resources.

Emphasizes speedup, scalability, and coordination overhead.

- Phase IV: Four Processors with Resource Contention: Students share resources while following strict color-order constraints. A scheduler manages contention using a FIFO approach. Demonstrates file contention, mutual exclusion, and resource sharing challenges.

Table 4.9 summarizes the approximate timing of each component of the activity implemented during a 75-minute class session. The class opened with a brief framing of the day's goals and a pre-activity quiz to establish a baseline measure of students' prior knowledge, followed by a short conceptual introduction to PDC and the vocabulary central to the activity. The bulk of the period was devoted to the unplugged activity itself, in which student groups executed four decomposition scenarios of increasing parallelism, recorded the completion time of each scenario. The session closed with a post-activity quiz and an engagement survey. For more details, please refer to Chapter 1, Section 1.3.1.

Table 4.9: USI CS1 Approximate Timing of Flag Maker Activity

Activity	**Duration**
Welcome and learning goals	3 min
Step 1: Pre-Test	8 min
Introduction: What is parallel computing? "Digging Holes" video	5 min
PDC glossary: data/task parallelism, sequential vs. parallel computing, multicore, multiprocessor, synchronization, scheduler, race condition	8 min
Step 2 setup: Activity overview, coloring rules (correct vs. incorrect), distribution of flag sheets and markers, group formation	5 min
Scenario 1: Single processor (64 cells; one colorer plus timer); record time	7 min
Scenario 2: Two processors (∼4 min run) and discussion (∼3 min)	7 min
Scenario 3: Four processors by stripe (∼3 min run) and discussion (∼3 min)	6 min
Scenario 4: Four processors by column (∼3 min run) and discussion (∼4 min)	7 min
Whole-class debrief: comparison of recorded times, speedup, load balancing, decomposition strategy	6 min
Step 3: Post-Test	7 min
Step 4: Post-activity student engagement survey	5 min
Closing and takeaways	1 min
Total	**75 min**

4.3.1.3 Experience and Teaching Tips

The Flag Maker activity provided a positive teaching experience by demonstrating that complex PDC concepts can be effectively introduced in an introductory computer science course through active, unplugged learning. Survey results indicated high levels of student engagement, enjoyment, collaboration, and interest in parallel computing, with many students reporting that the activities helped clarify misconceptions about parallelism and the relationship between processor count and performance. Although some students suggested improvements such as incorporating code examples, increasing activity variety, and refining the classroom setup, the overall experience showed that the activity successfully promoted understanding of key PDC concepts and provided an accessible framework for instructors seeking to integrate PDC topics into CS1 curricula.

Based on student feedback, assessment findings, and lessons learned from implementing the Flag Maker, several improvements could enhance future teaching sessions. First, incorporating visual aids, code snippets, and simplified programming examples alongside the unplugged activities could help students connect conceptual understanding with actual parallel programming practices, particularly for students seeking a deeper technical perspective. Second, rotating student roles (e.g., processor, scheduler, observer) throughout the activities helps increase participation and ensure that all students experience multiple perspectives of parallel systems. Third, providing students with a brief concept guide or terminology handout before the activity may improve comprehension and reduce confusion during execution. Additionally, to strengthen reflection and knowledge retention, instructors could incorporate short debriefing discussions or reflection exercises after each activity phase.

4.3.1.4 Data and Analysis

The data collection in this study was done via (i) single group pre/post test to measure student learning, where each student was graded on six PDC multiple choice questions testing their knowledge on sequential processing, parallel processing, speedup, contention, scalability, and race condition. (ii) A post activity survey was collected from participants to measure student engagement.

Pre/Post Test Analysis:
The pre/post test analysis demonstrates a substantial improvement in student understanding of parallel computing concepts. Among the 24 students who completed both assessments, the average score increased from 3.67 to 5.13 out of a total score of 6, representing a mean gain of 1.46 points. Nineteen students improved their scores, five showed no change, and no students performed worse on the post-test. Statistical analysis confirmed that the improvement was highly significant (Wilcoxon signed-rank test, $p < .001$; paired t-test, $p < .001$), indicating a meaningful learning impact. The greatest improvements occurred on concepts related to multi-processor drawbacks and speedup, while gains were observed across nearly all PDC concepts. Overall, the results suggest that the FlagMaker intervention was highly effective in increasing students' understanding of fundamental PDC concepts and produced substantial learning gains without any decline in student performance. Figure 4.1 illustrates the comparison of the percentage of correct answers for each of the six questions in the pre-

and post-tests.

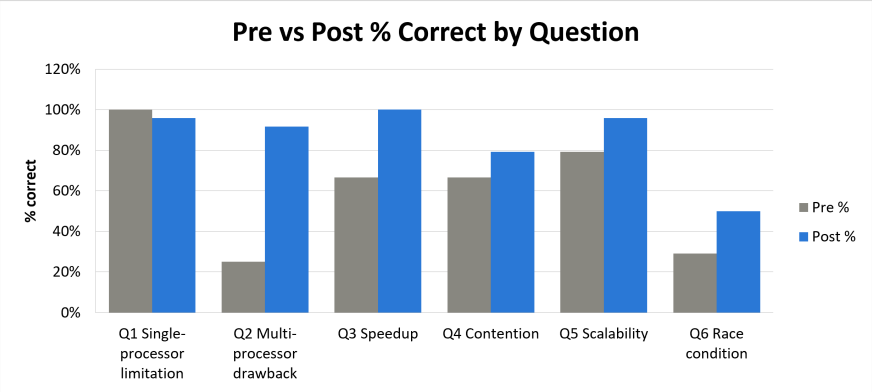


Figure 4.1: USI CS1 Pre vs Post Percentage of Correct Questions for N=24

Post Activity Student Engagement Survey Analysis:
Student engagement with the FlagMaker activity was strongly positive. The survey contained 18 five-point Likert items (Strongly Disagree = 1 to Strongly Agree = 5), two open-text questions (Q2: most interesting thing learned; Q3: suggested improvements), and optional demographics and was filled out by 27 students. Across the 18 Likert items the overall mean was 4.37 out of 5, with 16 items at 85% agreement or higher. Instructor-related items and peer/group items scored highest, and students reported enjoying the activity and seeing it as relevant to current course topics. One item stands clearly apart: "The activity increased my understanding of loops" (mean 3.48, 52% agreement, 33% neutral) was the only item to draw any disagreement and is the single notable weak spot. The diverging bar chart in Figure 4.2 below shows the distribution of responses for each item, centered on neutral, sorted from highest to lowest mean. Bars extending right indicate agreement; the few extending left indicate disagreement. Item mean is shown at right in the figure.

In the open response questions, students most often named contention, core parallelism concepts, and the diminishing returns of adding processors as their most interesting takeaways, which aligns with the concepts that improved most on the paired pre/post test. Suggestions for improvement were dominated by satisfaction ("nothing to change") or by the improvements in physical logistics of the coloring task (paper size, table space, markers). Responses to the two open questions were coded into the themes as shown in Figure 4.3 along with the theme frequencies.

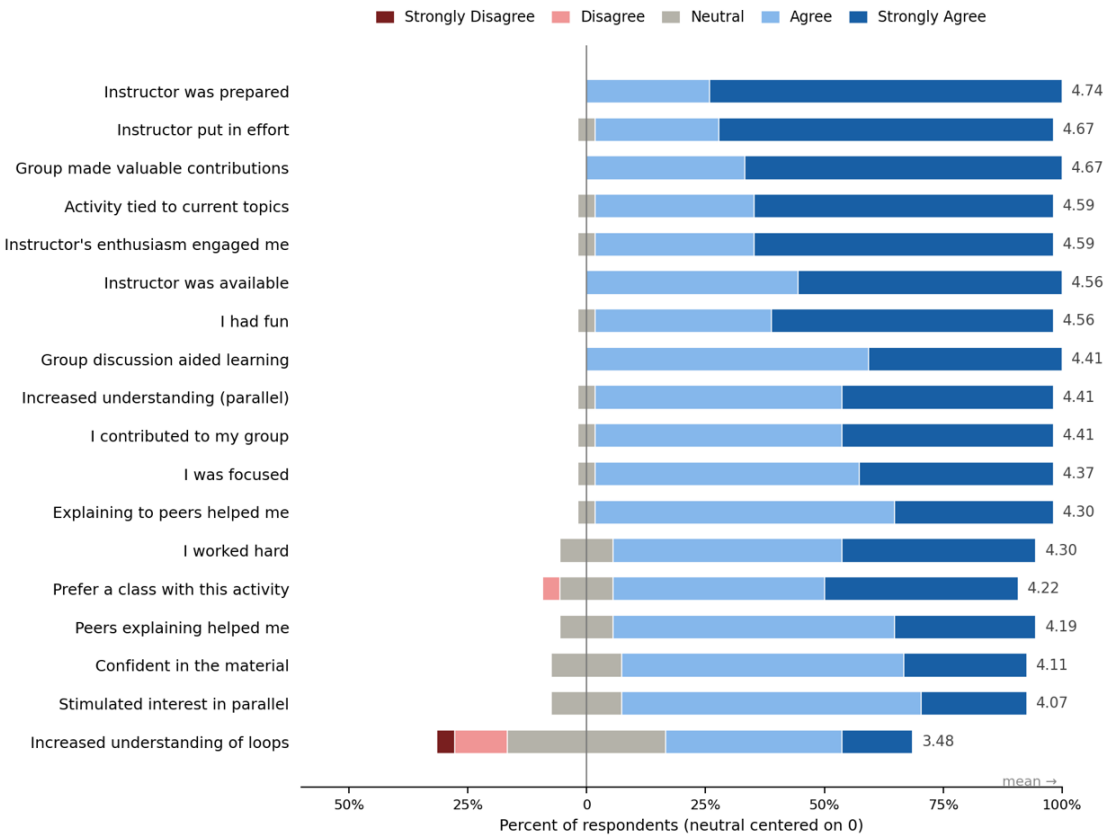


Figure 4.2: USI CS1 Response Distribution by Item

4.3.2 Unplugged Penny Sorting Exercise

4.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

The Unplugged Penny Sorting Search Exercise (PE) is a hands-on learning activity designed for CS1/CS2 students, requiring no prior parallel computing background beyond basic algorithmic thinking. The problem description centers on the physical task of sorting and searching through a collection of pennies to identify the oldest coin, using this tangible exercise to model processor roles. The activity's learning objectives are to help students grasp data parallelism, distinguish between load-balanced and load-imbalanced workloads, understand how the slowest processor bounds execution time, explain Amdahl's law, recognize communication overhead, and compare theoretical vs. observed speedups. For a more detailed description of the activity, please refer to Chapter 1, Section 1.3.2.

4.3.2.2 Implementation Details

The material used for implementing this activity can be found at the following GitHub link: https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter4-USI/Unplugged_Activities/PENNY_SORTING. This activity is a self-contained instructional module structured across four main phases:

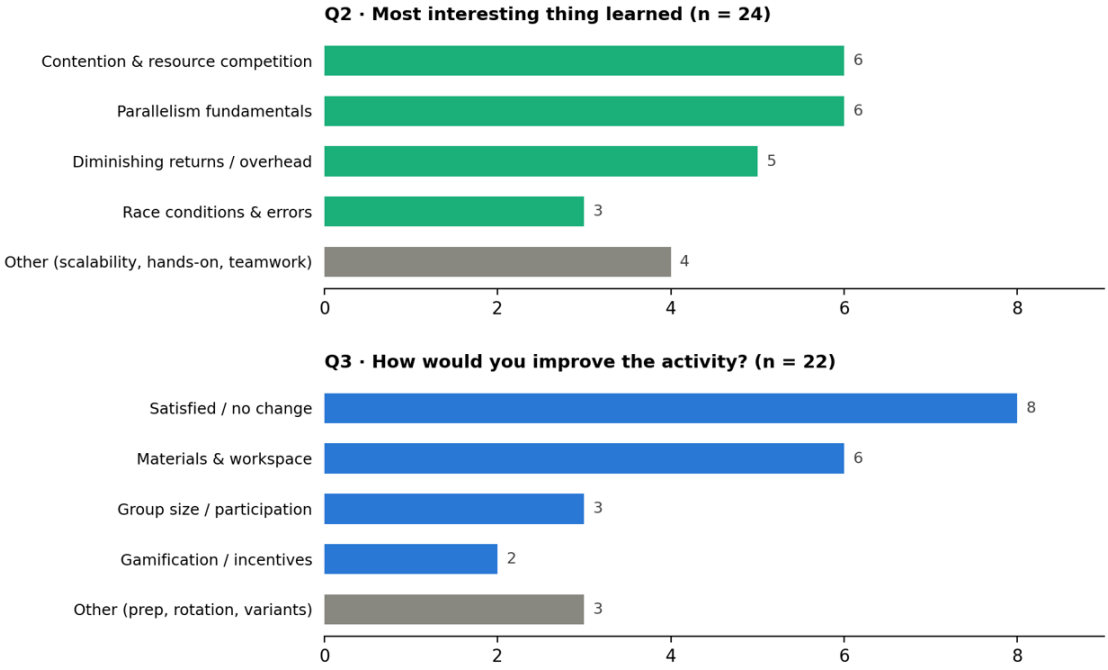


Figure 4.3: USI CS1 Thematic Analysis of Open Response Survey Questions

- a single-processor baseline (one student sorts, one times),
- a balanced two-processor parallel execution (two students split pennies equally, one times),
- a highly imbalanced five-processor parallel scenario (four students get two pennies each, while the fifth handles the remaining majority), and
- a final parallel execution time bounded by the slowest student plus communication overhead, leading into a broader instructional arc that concludes with video resources, structured group reflection, and an optional feedback survey.

For detailed step-by-step instructions on how to implement this activity, please refer to Chapter 2, Section 1.3.2.

Table 4.10 summarizes the approximate timing of each component of the penny sorting session. The class opened with a brief framing of the activity and its connection to sorting, a topic students had encountered earlier in the course, followed by group formation and distribution of materials. The bulk of the period was devoted to the unplugged activity itself, in which student groups searched a pile of pennies for the oldest coin under three successive work-distribution phases: sequential, evenly parallelized, and deliberately unbalanced, and recorded the completion time of each phase. Students then consolidated their results in a group discussion before completing an engagement survey.

4.3.2.3 Experience and Teaching Tips

The Unplugged Penny Sorting Exercise effectively fits within a standard 50- to 75-minute class, dedicating 20 to 25 minutes to the active phases and the remainder to discovery-based discussion. Pedagogically, the activity relies on firsthand experience to build intuition: the

Table 4.10: USI CS1 Approximate Timing of Unplugged Penny Sorting Exercise

Activity	**Duration**
Introduction: framing the activity, connection to sorting covered in class, goal of finding the oldest penny	4 min
Activity overview and setup: group formation (groups of 5–6), role assignment, distribution of coin bags	7 min
Phase 1 (sequential): one student searches all 50 pennies, a second times the search, remaining students observe; record time, shuffle, and reset	16 min
Phase 2 (parallel, balanced): two students each search 25 pennies, a third records both times; parallel time taken as that of the slower student; shuffle and reset	13 min
Phase 3 (parallel, unbalanced): four students receive two pennies each and a fifth receives the remainder, with one student timing; parallel time taken as that of the slowest student; shuffle and reset	12 min
Group discussion: what aspects of parallel computing did we observe?	6 min
Whole-class debrief and questions: comparison of recorded times, speedup, and load balancing	9 min
Post-activity student engagement survey	5 min
Closing and takeaways	3 min
Total	**75 min**

uniprocessor phase establishes a slow baseline, Phase 2 reveals tangible parallel speedup, and Phase 3 introduces a stark load imbalance where underutilized students sit idle, offering a perfect entry point to discuss processor starvation. Real-world variance in student sorting and communication times should be embraced as a teaching tool to illustrate how overhead prevents real systems from reaching perfect theoretical speedups. To scale the activity, instructors can use teaching assistants to monitor group mechanics in larger classes or pivot to a front-of-room volunteer demonstration for massive lecture halls. Ultimately, the exercise is most impacting when treated as a discovery experience, allowing students to analyze their own timing data and intuitively derive parallel computing concepts.

4.3.2.4 Data and Analysis

The participating students completed an anonymous engagement survey administered through Google Forms that contained Likert-scale items that measure student engagement, each rated on a five-point scale from 1 (Strongly disagree) to 5 (Strongly agree). The student engagement survey was adapted from the *Assessing Student Perspective of Engagement in Class Tool (ASPECT)* [69]. Several survey items were modified to reference the PDC activity and its learning objectives, and two PDC-specific questions were added to evaluate outcomes unique to this instructional intervention. The survey retained the three primary ASPECT categories: *Value of the Group Activity*, *Personal Effort*, and *Instructor Contribution*. Figure 1.3 illustrates the modified version of the ASPECT survey that was used for measuring student engagement for this activity. A total of thirty three students responded. Engagement items were organized into four dimensions: behavioral (effort, focus, participation), emotional (enjoyment, interest), cognitive (understanding and sense-making), and social (peer contribution). Two categories of items were set aside from the core engagement analysis: a baseline item measuring prior PDC knowledge, and four items measuring instructor delivery (preparedness, effort, availability), which reflect teaching quality rather than student engagement.

Engagement was strongly positive across the board. The average of the engagement item means is approximately 4.15 on the 5-point scale, and most items reached agreement (a

rating of 4 or 5) from more than 80% of respondents. Figure 4.4 presents a diverging stacked bar chart, where each bar is centered on the neutral response (3), so disagreement extends left and agreement extends right. Bars are centered on the neutral response; item-label color denotes engagement dimension. The percentage agreeing (ratings of 4 or 5) is printed at the right of each bar, and item labels are colored by engagement dimension. Emotional and social items cluster at the far right with little to no left-side disagreement, while the three peer-learning cognitive items (orange) carry visibly more neutral and disagreeing responses.

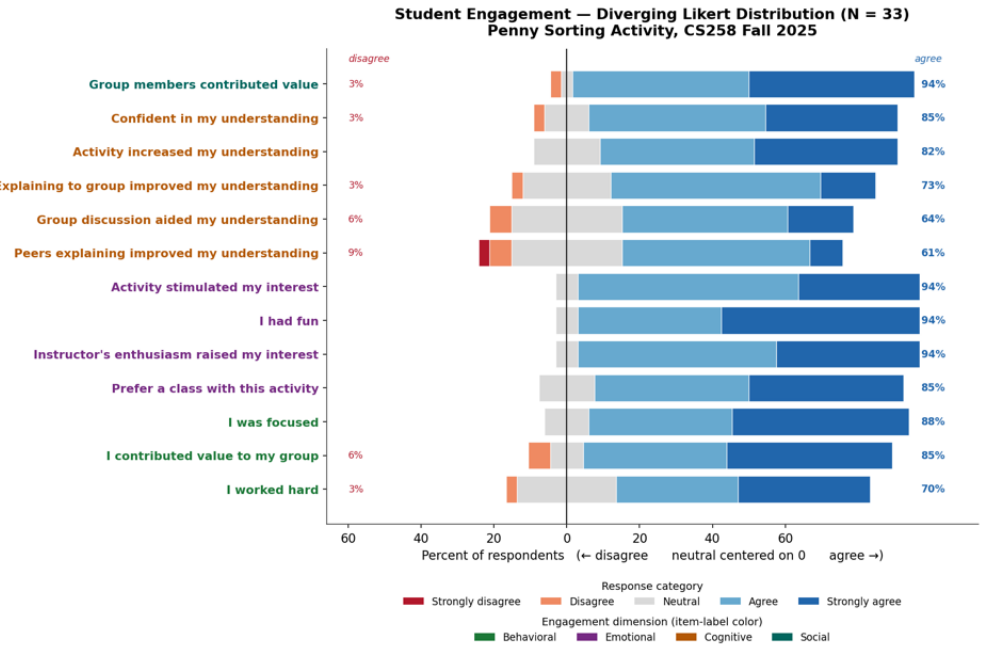


Figure 4.4: USI CS1 Diverging Likert Distribution of Engagement Items (n = 33)

Overall, the Unplugged Penny Sorting Exercise generated high engagement, driven most strongly by enjoyment, interest, and positive peer dynamics. The baseline prior-knowledge item was low (mean 2.67), which is expected and appropriate as it indicates that students entered with limited PDC familiarity, so the activity was introducing genuinely new material. The main actionable signal is the softer response to peer-based sense-making. Students valued the activity's effect on their own understanding more than they valued explaining to, or learning from peers. In the future iterations, this can be addressed by structuring the group discussion component more deliberately.

4.4 New Plugged-in Activities

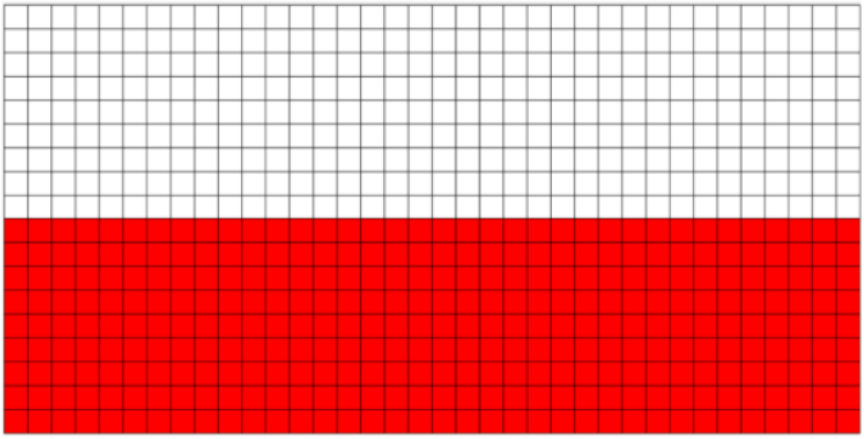
4.4.1 Flag Maker Plugged-In Activity

4.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

The Flag Maker Plugged-In Activity is a programming assignment in which CS1 students design and implement algorithms to draw a series of international flags using a grid-based

graphical environment. Although this activity is described as a PDC intervention for a CS1 course, it is inherently a sequential implementation of flag coloring. However, completing this activity before the parallel, Unplugged Flag Maker Activity prepared the students to better understand workload distribution when coloring a flag. Students are required to analyze each flag's geometric patterns, proportions, colors, and stripe layouts, then develop Java code using loops, conditional logic, variables, and coordinate-based grid manipulation to accurately generate scalable flag designs of varying sizes. The assignment includes flags such as Poland, Ukraine, Angola, Bolivia, Ireland, Rwanda, India, and a rainbow flag, with specific mathematical constraints related to height, width, and color distribution. An example of a gridded flag layout and corresponding code sample is illustrated in Figure 4.5.

Example when grid is size 18



Code sample:

```
int w = grid.getNumCols();
int h = grid.getNumRows();
for (int c=0; c < w; c++) {
    for (int r=h/2; r < h; r++) {
    grid.setColor(r, c, Color.RED);
    }
}
```

Figure 4.5: USI CS1 Example Flag Represented as a Colored Grid and Corresponding Code

As a prerequisite, to successfully complete the activity, students should possess foundational programming knowledge including variables, arithmetic expressions, nested loops, conditional statements, method structure, object-oriented concepts, and basic use of graphical grid APIs. They must also be able to reason about decomposition, coordinate systems, and proportional calculations to translate visual patterns into algorithmic solutions. The project assumes familiarity with Java development tools such as VS Code and the provided project framework.

Upon successful completion of this activity, students will be able to:

- Identify fundamental programming constructs including variables, loops, conditionals, methods, and grid-based graphical operations used in Java programs.
- Explain how national flag designs can be decomposed into geometric patterns, color regions, and proportional relationships that can be represented algorithmically.
- Implement nested loops, arithmetic calculations, and conditional logic to generate scalable flag designs that conform to specified dimensions and color arrangements.
- Test and verify that flag-generation algorithms produce correct output for multiple grid sizes rather than only the provided examples, identifying and correcting logical errors as needed.

4.4.1.2 Implementation Details

The material used for implementing this activity can be found at the following GitHub link. This activity was implemented as an individual programming assignment and served as a follow-up application of concepts learned in the course module on loops and iterative problem solving. Students were provided with a Java-based project framework and asked to complete a series of functions that generated multiple international flag designs using a grid-based graphical environment. The assignment required students to use nested loops, arithmetic calculations, variables, and conditional logic to color specific regions of a grid according to the proportions and patterns of each flag. Students were expected to create scalable solutions that worked correctly for different grid sizes rather than hard-coded dimensions. The assignment included progressively complex flag designs that required students to reason about proportions, coordinate systems, and pattern generation using loops.

Table 4.11 summarizes the timing of the Flag Maker Plugged-In Activity. The project was introduced during the first ten minutes of a regular 75-minute class session, in which the instructor framed the task, demonstrated the provided Poland flag example at several grid sizes, walked through the supporting code, and covered setup and submission logistics. Students then completed the project in pairs over the following week, drawing seven additional flags of increasing structural complexity. Because the graphical interface and one worked example were supplied, student effort was directed toward decomposing each flag into row and column ranges rather than toward interface programming.

This activity was completed before students were introduced to the Unplugged Flag Maker Activity. As a result, students initially approached the problem using sequential programming techniques and loop-based algorithms. Their experience developing solutions for drawing flags in code provided a foundation for later discussions about task decomposi-

Table 4.11: USI CS1 Approximate Timing of the Flag Maker Plugged-In Activity

Activity	Duration
Introduction to the project	2 min
Live demonstration of the provided Poland flag example, run at several grid sizes	3 min
Walkthrough of the provided code: the pre-coded GUI, `FlagMaker.java`, and the nested-loop sample that renders Poland	2 min
Project setup instructions: workspace file, selection of the platform-specific required Java extensions, and common pitfalls	2 min
Assignment logistics: pair formation, one-week deadline, grading rubric, and submission procedure	1 min
Regular class lecture: iteration (loops) in programming	65 min
Total (class period)	**75 min**
Out-of-class paired programming: environment setup, implementation of the remaining seven flags, testing at multiple grid sizes, and submission	1 week

tion, workload distribution, and parallel execution in the unplugged activity, where similar flag-design problems were explored from a parallel computing perspective. This sequencing allowed students to first master the underlying algorithmic concepts before extending their understanding to parallel problem-solving approaches.

4.4.1.3 Experience and Teaching Tips

The assignment required students to analyze the mathematical structure of each flag, decompose complex visual designs into simpler programming tasks, and develop scalable solutions that worked for multiple grid sizes rather than fixed examples. Through hands-on coding, testing, and debugging, students gained practical experience applying fundamental programming concepts to solve visually engaging problems, while also strengthening computational thinking, problem decomposition, and collaborative software development skills. Allowing the students to work on the activity as a paired programming assignment improved the interest levels and enhanced student learning.

4.4.1.4 Data and Analysis

No data was collected in the form of surveys or pre/post test. The learning was assessed by grading the programming assignment that the students submitted. A total of 37 out of 39 students completed the assignment. Each student successfully completed at least 8 out of the 10 flags provided in the assignment. Overall, the assignment seemed to improve the understanding of the concepts of loops and iterative problem solving for students. This assignment also helped the students when they worked on the unplugged activity as they could easily translate their coding experience to the on-paper experience.

4.4.2 Parallel Penny Sort Plugged-In Activity

4.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

In this activity, CS1 students were tasked with developing a Java application that simulates sorting a collection of pennies based on their minted years. Students generated an

array of 50 random penny years, sorted the data using both a sequential sorting method and Java's parallel sorting functionality, measured the execution times of each approach, and identified the oldest penny in the collection. The project also encouraged students to analyze and compare sorting performance and, for extra credit, implement a binary search to locate a user-specified penny year within the sorted array.

As a prerequisite, before completing this activity, students were expected to have foundational knowledge of Java programming, including variables, control structures (loops and conditionals), methods, arrays, and basic algorithmic problem solving. Students also needed familiarity with generating random values, traversing arrays, and writing modular programs. Understanding of sorting concepts and basic program testing techniques was beneficial for successfully implementing and evaluating the required solutions. The activity served as a culminating project following instruction on arrays and introductory algorithmic techniques.

Regarding learning outcomes, upon completing this activity, students were expected to:

- Store and manipulate integer data using Java arrays.
- Implement and evaluate sorting algorithms using both sequential and parallel approaches.
- Generate and process randomized datasets for program testing and validation.
- Develop algorithms to locate the minimum value within a dataset and identify the oldest penny.
- Measure, compare, and analyze algorithm performance using execution-time data collected over multiple runs.
- Apply software engineering practices such as modular design, method decomposition, documentation, and testing to create readable and maintainable code.
- (Optional) Implement binary search on a sorted array and interpret search results.

4.4.2.2 Implementation Details

The material used for implementing this activity can be found at the following GitHub link. This activity was implemented as a team-based programming assignment following the Unplugged Penny Sorting Exercise. Students designed, developed, tested, and analyzed a Java application that worked with arrays of penny mint years. Students generated an array of exactly 50 randomly produced years representing pennies, displayed the unsorted data, and then applied two different sorting approaches: a sequential sorting method (such as a manually implemented bubble, insertion, or selection sort, or Arrays.sort) and Java's built-in Arrays.parallelSort method. Students measured and compared the execution times of both sorting techniques across multiple runs and reported average, minimum, and maximum execution times. After sorting the data, students implemented a single-pass algorithm to identify the oldest penny (minimum year) in the collection. The project emphasized modular programming, testing, code documentation, and performance analysis. As an optional enhancement, students could implement a binary search algorithm to locate a user-specified year within the sorted array and report whether the value was found.
Here are the instructions that were provided to the students in the assignment documentation:

1. Generate an array of 50 integer values representing penny mint years using a random

number generator.
2. Use Java arrays (int[]) as the primary data structure and avoid using ArrayList or streams for the core logic.
3. Display the original unsorted array of penny years.
4. Implement a sequential sorting solution using a manual sorting algorithm (e.g., selection, insertion, or bubble sort) or Arrays.sort.
5. Implement a parallel sorting solution using Arrays.parallelSort.
6. Measure and record the execution time for both sorting approaches.
7. Execute each sorting approach multiple times and calculate summary statistics such as average, minimum, and maximum run times.
8. Sort the array in ascending order so that the oldest penny appears first.
9. Print the sorted array, the oldest year found, and a summary of the timing analysis.
10. (Optional Extra Credit) Implement a binary search method that allows a user to search for a specific penny year and report its location or indicate that it was not found.
11. Submit a zipped folder containing source code, a README file, presentation materials, and analysis of the sorting algorithms.
12. Demonstrate the completed project and present findings during the final project presentation.

Table 4.12 summarizes the timing of theParallel Penny Sort Plugged-In Activity. The project was introduced during a regular 75-minute class session that opened with team formation, followed by a description of the problem statement and requirements and an extended Q&A period. Students spent the remainder of the session working within their teams to decompose the project into distinct tasks, assign ownership, and raise final questions with the instructor present. Teams then completed the implementation over the following week, timing a sequential and a parallel sort of the same data and analyzing the results, with a code demonstration and presentation due on the final exam day.

Table 4.12: USI CS1 Approximate Timing of the Penny Sort Plugged-In Activity

Activity	**Duration**
Team assignment: formation of project teams and distribution of the project instructions	5 min
Project description: problem statement, learning objectives, sequential and parallel sorting requirements, timing and analysis expectations, deliverables, rubric, and teamwork guidelines	10 min
Question-and-answer session on project requirements and expectations	10 min
In-class team work: task breakdown and assignment among team members, setup of the project management board, planning of the timing analysis, and follow-up questions to the instructor	50 min
Total (class period)	**75 min**
Out-of-class team programming: implementation of the sequential and parallel sorts, repeated timing runs, analysis, documentation, and preparation of the demonstration and presentation	1 week

4.4.2.3 Experience and Teaching Tips

A primary challenge for students was implementing the parallel sorting method and verifying that it accurately ordered the data from oldest to newest. To achieve a correct

implementation, students had to research parallel sorting techniques, grasp their underlying mechanics, and understand their overall functionality. Working in teams greatly facilitated this research process, as group discussions helped most students successfully conceptualize parallel sorting.

For the extra credit component, students had to recognize the prerequisite that a binary search only functions on sorted data, and then correctly implement logic to return either the matching year's index or a "not found" result. This added complexity deterred many students from attempting the optional section. In future iterations, however, this search component could be integrated into the activity as a core requirement.

4.4.2.4 Data and Analysis

This activity was assessed by grading the programming assignments submitted by student teams and evaluating their performance during an in-class presentation. During their presentations, students explained their research into parallel sorting algorithms and connected their coding work back to the unplugged penny sorting activity, noting that the hands-on exercise significantly helped them program the parallel concepts. To improve this assessment in future iterations, the evaluation could be enhanced by introducing a structured grading rubric for the presentation, incorporating a peer-evaluation component to track individual contributions within teams, and utilizing a post-activity reflection survey to more objectively capture student learning gains.

4.4.3 Greenfoot Plugged-In Demo (time permitting/optional for CS1)

4.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

As undergraduate computer science students transition from CS1 to CS2, they often face a steep learning curve when shifting from purely sequential, console-based programming to complex, non-linear execution models. Introducing advanced PDC paradigms like event-driven programming concepts too abruptly can overwhelm learners. To bridge this pedagogical gap, a specialized module was designed for the conclusion of the Fall 2025 CS1 course. The goal was to provide a visual, intuitive, and low-stakes sneak peek into CS2 fundamentals, demonstrating how asynchronous events interrupt sequential execution and map to real-world software interactions.

Students entering this module were required to have a foundational understanding of core CS1 concepts, including basic Java syntax, control structures (loops and conditionals), variables, and rudimentary object-oriented principles such as classes and methods.

The primary learning outcomes for this introductory module were for students to be able to:

- Identify the core differences between sequential execution and event-driven execution models.
- Describe how a graphical environment listens for, captures, and responds to asynchronous user inputs (e.g., keyboard and mouse events).

- Recognize the conceptual alignment between event handling and broader concurrent or parallel computing paradigms, laying a conceptual foundation for future CS2 coursework.

4.4.3.2 Implementation Details

A number of Greenfoot activities can be found on their website at: `https://www.greenfoot.org/home`. The module was implemented using Greenfoot [34], an educational visual Java development environment that combines an interactive object-oriented world with standard Java code. Toward the end of the Fall 2025 semester, a live interactive demonstration was conducted. The demo utilized a simple 2D simulation where "Actor" objects reacted in real-time to user interrupts. By examining the environment's underlying execution loop (the `act()` method) alongside keyboard listeners (such as `Greenfoot.isKeyDown()`), the mechanics of event loops and event handlers were explicitly deconstructed. This visual feedback allowed students to see how real-time user inputs break the traditional top-down flow of execution, introducing them to the asynchronous nature of graphical user interfaces and concurrent systems.

The instructional delivery relied on a blend of synchronous demonstration and supportive multimedia resources, including:

- Interactive Lecture and Live Coding: A guided session introducing the Greenfoot interface, contrasting a traditional console-based Java program with an event-driven graphical environment.
- Curated Video Tutorials: Short, targeted video walkthrough from the Greenfoot repository were provided to students. These videos reinforced how the platform manages spatial coordination, actor interaction, and background event listeners, allowing students to revisit the mechanics at their own pace.

Table 4.13 summarizes the approximate timing of the Greenfoot session, conducted at the end of the Fall 2025 semester.

Table 4.13: USI CS1 Approximate Timing of the Greenfoot Plugged-In Demo

Activity	**Duration**
Introduction and framing: motivation for event-driven execution, contrast between a traditional console-based Java program and an interactive graphical environment	5 min
Greenfoot demonstration and live coding: tour of the Greenfoot interface and a 2D simulation showing mechanics of event loops and event handlers	20 min
Student hands-on exploration: students modify actor behavior and experiment with real-time keyboard input, observing how user events break top-down flow of execution; curated video tutorials from the Greenfoot repository available for self-paced review, with the instructor present for questions	45 min
Closing discussion and questions: the asynchronous nature of graphical user interfaces, connection to concurrent systems, and semester wrap-up	5 min
Total	**75 min**

4.4.3.3 Experience and Teaching Tips

Utilizing a highly visual, gamified environment successfully lowered the barrier to entry for a complex topic. Students demonstrated high engagement, noting that seeing an immediate graphical response to a keyboard event made the abstract concept of listeners and interrupts concrete.

Lessons Learned and Teaching Tips for Future Iterations:

- Emphasize the Execution Loop: Spend sufficient time explaining that a background engine is continuously polling for events. This prevents the common misconception that the code written inside an actor is running in isolation.
- Bridge Explicitly to CS2: Explicitly use the vocabulary of CS2 (such as interfaces, inheritance, and concurrency) during the demo so that the terminology feels familiar when students encounter it formally in the subsequent semester.

4.4.3.4 Data and Analysis

Because this module was executed strictly as an introductory demonstration and an active learning preview, no formal quantitative data or assessment metrics were collected during the Fall 2025 semester. Formal data collection and empirical analysis are planned for a follow-up semester in the CS2 course, where students will be writing their own event-driven programs and exploring explicit PDC components. Assessment methodologies for the CS2 implementation will include code analysis rubrics and qualitative surveys regarding event based programming concepts.

4.5 Course-wide Pre and Post Course Surveys

To evaluate the impact of integrating PDC concepts into the introductory computer science curriculum, a comparative data collection study was conducted across two consecutive fall semesters. In Fall 2024, baseline data was established under a traditional curriculum structure without any explicit PDC interventions. To assess student growth and conceptual shifts, identical pre- and post-surveys were administered during the first and final weeks of the semester, respectively. The survey content is described in more detail in Chapter 1, Section 1.5. In Fall 2025, the same pre- and post-survey methodology was employed to measure the efficacy of a targeted pedagogical intervention consisting of three distinct PDC modules. As described in the sections above, this intervention integrated two "unplugged" and two "plugged-in" activities, collectively exposing students to core concurrent computing concepts including data parallelism, parallel speedup, task scheduling, and load balancing, as well as a foundational demonstration of event-driven programming using the Greenfoot environment. A key structural difference between the two semesters is the primary programming language: C# was used in Fall 2024, whereas Java was adopted for Fall 2025. Consequently, the subsequent data analysis acknowledges that this shift in language may act as a confounding factor. Nevertheless, the overall results strongly indicate that the PDC intervention effectively improved students' self-reported understanding of parallel and distributed computing concepts.

4.5.1 Results and Analysis: Fall 2024 Benchmark Survey Data

Of 26 pre-course and 18 post-course unique respondents, 13 students completed both surveys and form the paired sample. The matched baseline cohort is small (13), split across freshman/sophomore/junior standing. Nearly all expected an A. An analysis of the pre/post survey results collected during Fall 2024 (baseline semester) is summarized in Table 4.14. Figure 4.6 shows the shifts in the Likert scale (7-point scale) values for the questions on the pre- and post-surveys. All 14 matched students answered the three free-text items.

Table 4.14: USI CS1 Pre/Post Survey Results from Benchmark Fall 2024

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Computing Background									
experience-computer	13	5.77	6.15	0.38	3	0.188	No	−0.17	small
experience-programming	13	4.15	4.54	0.38	6	0.250	No	−0.15	small
experience-peers	13	3.77	4.54	0.77	3.5	0.027	Yes	−0.26	small
Programming Concepts									
Data/Variable Types	13	4.15	5.23	1.08	7	0.021	Yes	−0.25	small
Arithmetic Expr. and Operators	13	4.31	5.23	0.92	7	0.043	Yes	−0.19	small
Branching	13	4.38	5.08	0.69	6	0.133	No	−0.14	negligible
Loops	13	4.23	4.77	0.54	14	0.363	No	−0.08	negligible
Calling functions/methods	13	4.31	4.92	0.62	6	0.125	No	−0.14	negligible
Arrays	13	3.46	4.69	1.23	7	0.043	Yes	−0.30	small
Writing static functions	13	3.38	4.85	1.46	3	0.006	Yes	−0.44	medium
Writing methods	13	3.92	5.00	1.08	7	0.045	Yes	−0.26	small
Writing classes	13	3.46	5.00	1.54	3	0.012	Yes	−0.41	medium
Creating objects	13	3.54	4.85	1.31	5	0.047	Yes	−0.33	small
PDC Concepts									
parallel	13	2.08	3.62	1.54	3.5	0.003	Yes	−0.59	large
distributed	13	2.00	3.08	1.08	3	0.023	Yes	−0.44	medium
event	13	3.08	3.77	0.69	16.5	0.169	No	−0.23	small
Computing Attitudes									
computing interest	13	6.15	6.31	0.15	6	0.531	No	−0.24	small
PDC interest	13	5.77	5.85	0.08	16.5	0.961	No	−0.03	negligible
PDC importance	13	5.85	6.15	0.31	5.5	0.406	No	−0.22	small
computing understanding	13	4.77	5.31	0.54	3.5	0.109	No	−0.17	small
PDC understanding	13	3.46	4.08	0.62	13	0.289	No	−0.23	small

Pre-course aspirations: The clearest theme is that goals were framed around C#:

- student quotes on learning C# as a language (most common): "learning the C# syntax," "fully learning C#," "learn the syntax of C# as I've never used the language," "write high level programs in C#." One student tied it to C# based game engines.
- student quotes on general fundamentals: "different ways to think about programming," making code "more efficient," a "platform to begin enhancing my programming abilities."

What helped most: students credited hands-on, example-driven work, with peer collaboration and instructor quality:

- Quotes on code examples worked through in class (dominant): "coding examples in class were mostly how I learned C#," "examples of what the code should look like and why."

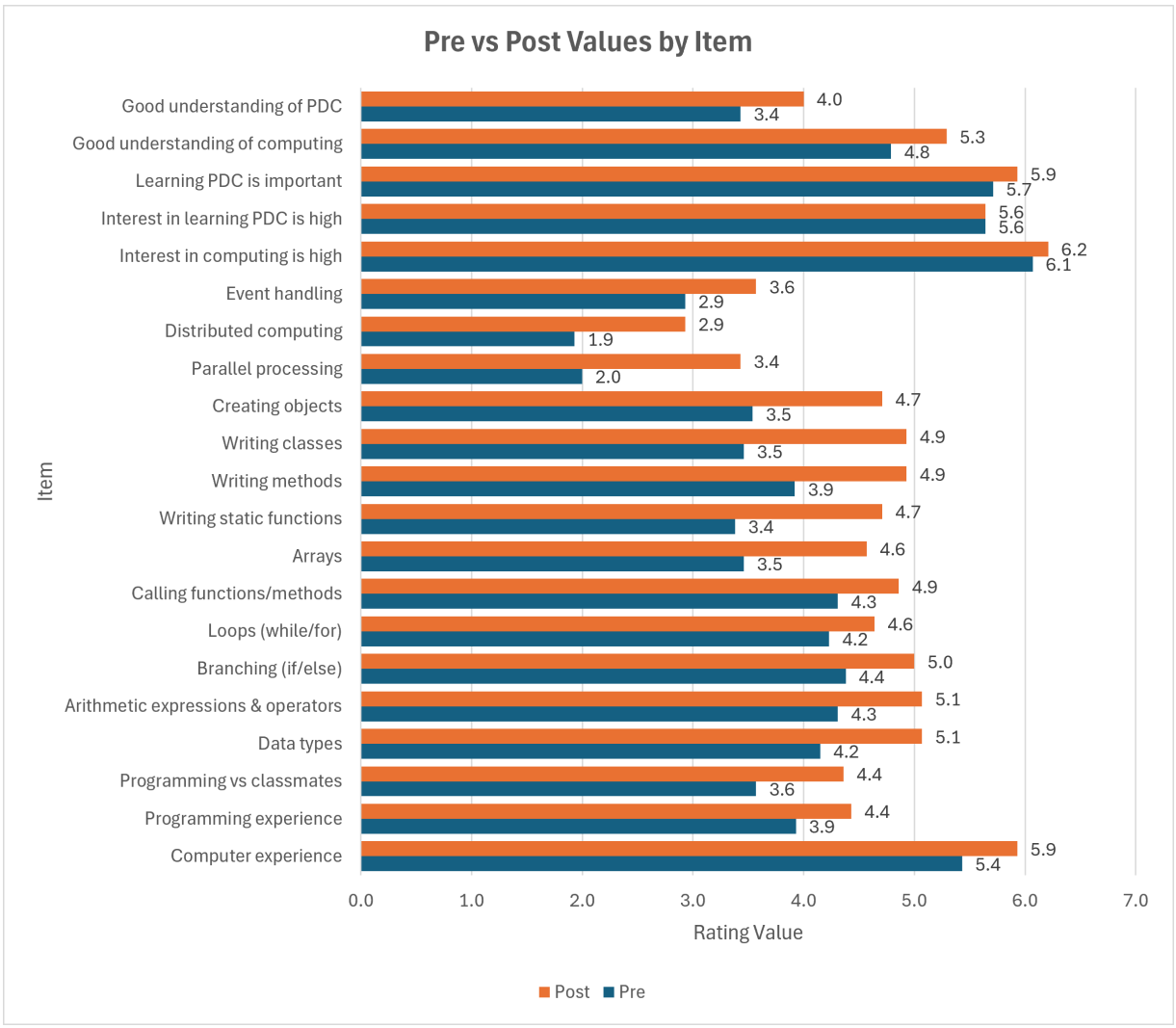


Figure 4.6: USI CS1 Fall 2024 Pre to Post Shift on Every Survey Item

- quotes on hands-on practice and homework: "programming assignments in lecture and in-class practices," "working on segments of the projects in class."
- quotes on peer collaboration and a caring instructor: "working with other people to problem solve," "getting help from others," "a professor that cares... and passionate."

Suggested adjustments: change requests centered on materials and assessment more than teaching style:

- quotes on textbook and materials: "the book could be optional," "a better textbook," "a more interactive way... to learn C#."
- quotes on exams and testing: "make exam questions less complex," more clarity on coverage, and one arguing tests "don't actually show you if you know how to do it."
- quotes from satisfied students: several saw little to change, one noted the class was "very easy" given their prior experience.

4.5.2 Results and Analysis: Fall 2025 PDC Intervention Survey Data

Of 41 pre-course and 27 post-course respondents, 25 students completed both surveys (2 students submitted partially complete surveys) and form the paired sample analyzed here. Ordinal responses were mapped to 1–7 numeric scales for effect-size estimation. The result demonstrate broad, large, statistically robust gains in programming and PDC experience, corroborated by the open-ended comments. The sample has predominantly Computer Science majors (26 of 27), largely traditional age (23 of 27 aged 18–22), and skewing early-undergraduate, though juniors and sophomores are well represented. Most students entered expecting an A or B. An analysis of the pre/post survey results collected during Fall 2025 (PDC intervention semester) is summarized in Table 4.15. Figure 4.7 shows the shifts in the Likert scale (7-point scale) values for the questions on the pre- and post-surveys.

Table 4.15: USI CS1 Pre/Post Survey Results from PDC Intervention Fall 2025

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Computing Background									
experience-computer	25	6.04	6.00	−0.04	22.5	1.000	No	0.03	negligible
experience-programming	25	3.48	4.68	1.20	6	0.000	Yes	−0.50	large
experience-peers	25	3.24	4.40	1.16	10	0.001	Yes	−0.43	medium
Programming Background									
Data/Variable Types	25	3.04	5.28	2.24	7.5	0.000	Yes	−0.61	large
Arithmetic Expr. and Operators	25	3.08	5.08	2.00	10	0.000	Yes	−0.60	large
Branching	25	3.40	5.32	1.92	8.5	0.000	Yes	−0.62	large
Loops	25	3.48	5.32	1.84	17.5	0.000	Yes	−0.58	large
Calling functions/methods	25	2.80	5.16	2.36	2.5	0.000	Yes	−0.62	large
Arrays	25	2.12	4.52	2.40	3	0.000	Yes	−0.66	large
Writing static functions	25	2.20	4.64	2.44	3.5	0.000	Yes	−0.72	large
Writing methods	25	2.16	4.88	2.72	0	0.000	Yes	−0.72	large
Writing classes	25	2.28	4.92	2.64	0	0.000	Yes	−0.70	large
Creating objects	25	2.36	5.08	2.72	0	0.000	Yes	−0.80	large
PDC Background									
parallel	25	1.76	4.12	2.36	0	0.000	Yes	−0.75	large
distributed	25	2.04	3.84	1.80	15.5	0.000	Yes	−0.56	large
event	25	3.32	4.32	1.00	20.5	0.013	Yes	−0.31	small
Computing Attitudes									
computing interest	25	6.36	6.24	−0.12	25	0.448	No	0.06	negligible
PDC interest	25	5.76	5.52	−0.24	44	0.341	No	0.16	small
PDC importance	25	5.68	6.12	0.44	32	0.052	No	−0.26	small
computing understanding	25	4.36	5.56	1.20	12	0.000	Yes	−0.46	medium
PDC understanding	25	3.24	5.40	2.16	4.5	0.000	Yes	−0.77	large

On nearly every construct the course targets, students moved roughly two full points on a seven-point scale between week 1 and the final week. The biggest jumps cluster in the object-oriented core : writing methods (+2.7), creating objects (+2.7), writing classes (+2.6), and in the PDC intervention (parallel processing +2.4, distributed computing +1.9). Four items did not shift significantly: computer experience, general interest in computing, interest in learning PDC, and (borderline) belief that PDC is important. All four were already high at pre-survey (means of 5.9, 6.4, 5.8, 5.7 on the 7-pt scale). Thus this can be regarded as a ceiling effect rather than a failure of the course.

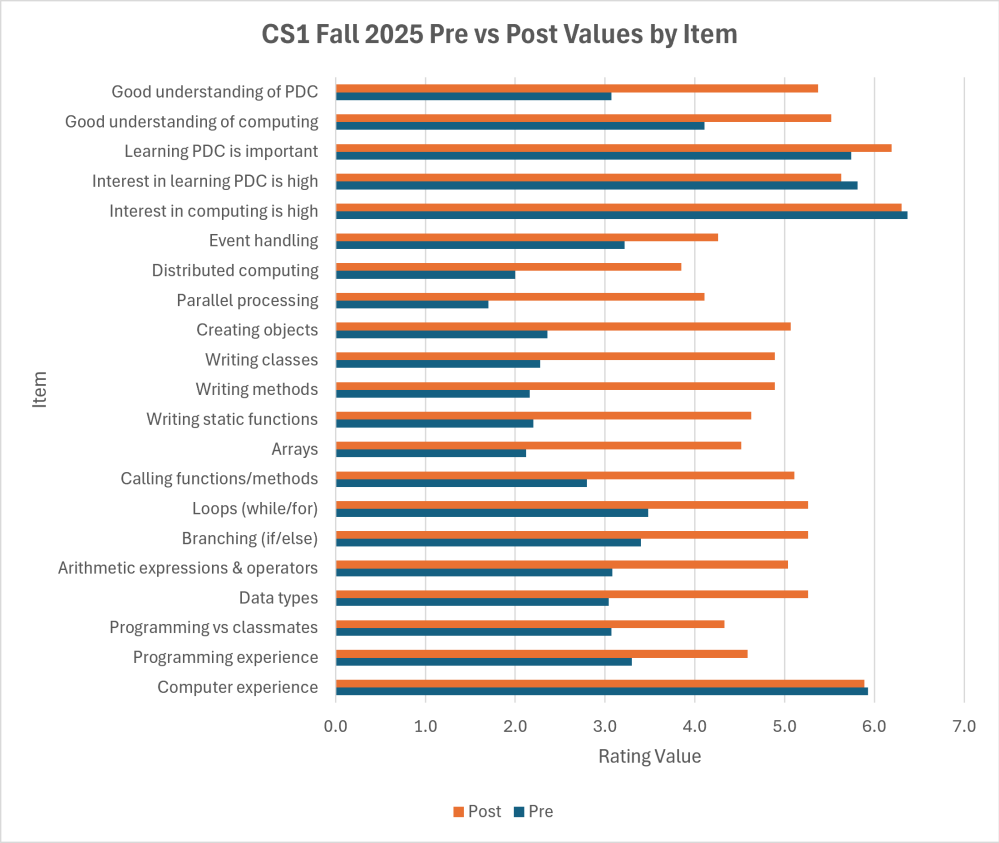


Figure 4.7: USI CS1 Fall 2025 Pre to Post Shift on Every Survey Item

Three text response items were coded thematically: pre-course aspirations (27 responses), and post-course "what helped most" (26) and "what to change" (20).

1. Pre-course aspirations: students entered with grounded goals rather than specialized ones. Four themes recurred:
    - Building basic fluency and confidence (most common): "be able to program myself," to grasp "basic coding fundamentals," or to get a "base start."
    - Java & object-oriented programming: understanding Java, "object based programming," better style and "optimization of code."
    - Career & real-world application: "my career path," "the real world," even "the enterprise level."
    - Conceptual understanding of how computers work: a reflective minority wanted the "why," not just the "how."
2. What helped most: a near-consensus for active learning
    - Hands-on, in-class coding & worked examples (20 of 26) — "in-class practices," "writing code in general," demos worked through live, and "seeing what the code is supposed to look like after I attempt my version."
    - Projects & from-scratch assignments: the midterm/final projects and "writing an entire code from scratch" were singled out as the deepest learning.

- The unplugged PDC activities: three students spontaneously praised the "unplugged activities" and the "penny sort" demonstration. This is direct evidence the successful PDC intervention.

3. Suggested adjustments: the change requests were constructive and mostly asked for more of what already worked:
   - More active coding, fewer slides (dominant): "more hands on doing the code ourselves than powerpoints,"
   - More projects & recurring mini-labs: including "mini labs every two-three weeks" to refresh earlier modules.
   - Accountability: some wanted in-class participation graded
   - Pacing & challenge: a couple felt the pace fast ("too much too quickly"). Others wanted more challenge. Several were fully satisfied.

4.5.3 Did the PDC Intervention Improve Learning?

To evaluate the impact of the intervention, a difference-in-differences (DiD) approach was utilized to compare the evaluation data between the Fall 2024 and Fall 2025 semesters. For each metric, the average learning gain (defined as post-survey score − pre-survey score) was computed for each cohort. The final treatment effect was estimated by subtracting the 2024 baseline gain from the 2025 intervention gain, isolating the marginal learning increase directly attributable to the PDC activities. A comparison of individual items between the Fall 2024 baseline and Fall 2025 intervention semesters is presented in Figure 4.8. On the composite measure of PDC concepts and self-rated understanding, the intervention cohort outperformed the baseline cohort in total points gained. Furthermore, the share of improving students increased from 79% to 93% in Fall 2025. These results suggest that the intervention nearly doubled self-reported PDC learning gains from a comparable baseline.

Figure 4.9 shows the specific PDC concept gains between the baseline cohort and the intervention cohort. Parallel processing and distributed computing experience each roughly doubled and are significant at the raw level. Event handling shows almost no difference and is consistent with it being integrated as a demo of the concepts the students will learn in CS2. One thing to note from these results is that concepts such as writing methods, calling functions, creating objects and others show large DiD values, but the 2024 cohort started higher on these and the teaching language differed between the cohorts (C# to Java). Overall, using Fall 2024 as a baseline, the evidence indicates that the PDC intervention improved students' self-reported learning of parallel and distributed computing concepts. From starting points that were statistically identical across cohorts, the intervention year produced roughly double the gain on PDC concepts. An analysis of the combined pre/post survey results collected during Fall 2024 (baseline semester) and Fall 2025 (PDC intervention semester) are summarized in Table 4.16.

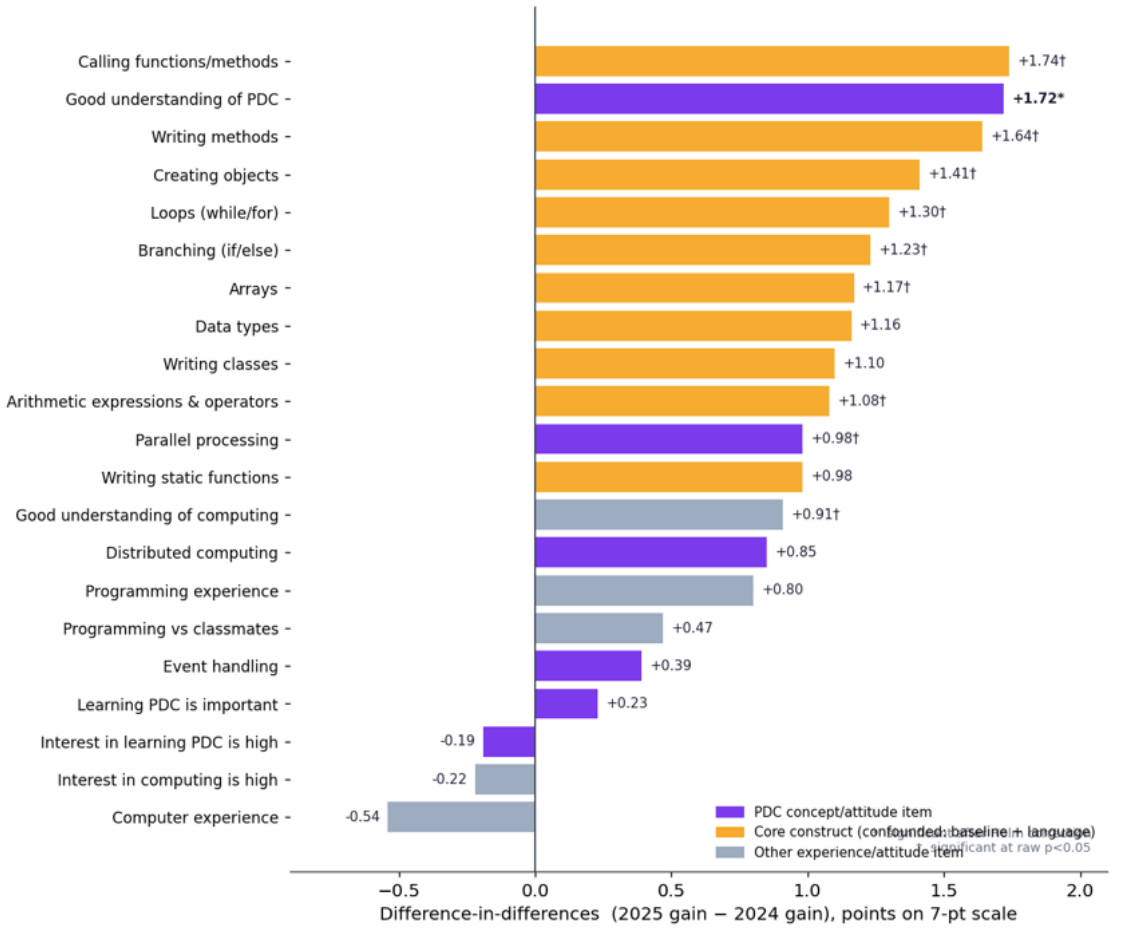


Figure 4.8: USI CS1 Improvement of Each Item from Benchmark Fall 2024 to PDC Intervention Fall 2025

4.6 Other Activities and Ideas

4.6.1 More Processors are Not Always the Best

The core problem addressed by this module is demonstrating how adding more hardware execution units (processors) does not inherently lead to a linear decrease in runtime. Instead, performance scaling is heavily constrained by dependencies, multi-tasking overhead, coordination delays, and structural imbalances. This active learning module is explicitly designed for an early undergraduate environment. Basic exposure to the concept of how a CPU executes a sequence of instructions sequentially. No prior parallel programming experience or specialized hardware access is required. The material used for implementing this activity in class can be found at the following GitHub link.

Learning Outcomes: By completing this unplugged activity, students should be able to:

- Differentiate between uniprocessor sequential execution, uniprocessor multitasking, and true multi-processor parallel execution.
- Explain how communication, coordination, and synchronization overhead can impede

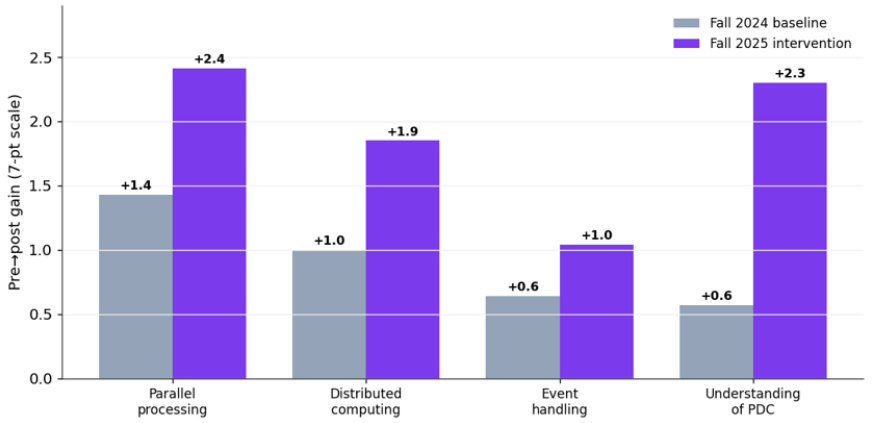


Figure 4.9: USI CS1 Pre to Post Gains on Four PDC Measures: Parallel Processing, Distributed Computing, and PDC Understanding

parallel efficiency.

- Identify execution dependencies that lead to idle processor time (e.g., waiting for data from an upstream process).
- Relate concrete physical execution blockages to abstract theoretical scaling laws, such as Amdahl's Law limitations.

Implementation Guidelines: the session is conducted in a standard lecture room by organizing students into collaborative groups of 5–6 members, where participants rotate through roles as individual "processors" or dedicated "timers". All execution processors are given their own local sheet of paper acting as an isolated local data buffer. The workload consists of two computational sub-tasks:

- Task 1: Writing out the string literal "more processors are not always the best".
- Task 2: Assigning a sequential index value (0, 1, ... 32) directly beneath each individual character of the written sentence.

Figure 4.10 shows the activity phases and how PDC concepts are implemented in those phases.

Table 4.17 summarizes the approximate timing of the unplugged "More Processors Are Not Always the Best" activity, implemented during a 75-minute class session.

Lessons learned and teaching tips: ensure students strictly work on their own physical sheets of paper. If they attempt to share a single piece of paper simultaneously, it will inadvertently simulate hardware memory bus contention rather than the intended message-passing synchronization overhead. Guide students to explicitly connect their physical experience of waiting for a teammate's indexing number to real-world parallel computing stalls and Amdahl's Law restrictions.

A survey similar to the ASPECT based survey used for the Penny Sorting activity discussed above was collected after this activity as well. Survey results indicated that the activity attributed to high levels of student engagement and therefore an improved interest in PDC learning. A more detailed description of the activity and an in-depth analysis of the student engagement results collected after the activity implementation can be found in [60].

Table 4.16: USI CS1 Combined Pre/Post Survey Results from Fall 2024 and Fall 2025

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Computing Background									
experience-computer	38	5.95	6.05	0.11	43	0.296	No	−0.03	negligible
experience-programming	38	3.71	4.63	0.92	25.5	0.000	Yes	−0.36	medium
experience-peers	38	3.42	4.45	1.03	24	0.000	Yes	−0.38	medium
Programming Concepts									
Data/Variable Types	38	3.42	5.26	1.84	27	0.000	Yes	−0.48	large
Arithmetic Expr. and Operators	38	3.50	5.13	1.63	29	0.000	Yes	−0.44	medium
Branching	38	3.74	5.24	1.50	24	0.000	Yes	−0.41	medium
Loops	38	3.74	5.13	1.39	57	0.000	Yes	−0.39	medium
Calling functions/methods	38	3.32	5.08	1.76	15	0.000	Yes	−0.46	medium
Arrays	38	2.58	4.58	2.00	18	0.000	Yes	−0.54	large
Writing static functions	38	2.61	4.71	2.11	12	0.000	Yes	−0.62	large
Writing methods	38	2.76	4.92	2.16	11	0.000	Yes	−0.56	large
Writing classes	38	2.68	4.95	2.26	5.5	0.000	Yes	−0.60	large
Creating objects	38	2.76	5.00	2.24	11	0.000	Yes	−0.64	large
PDC Concepts									
parallel	38	1.87	3.95	2.08	5	0.000	Yes	−0.70	large
distributed	38	2.03	3.58	1.55	28	0.000	Yes	−0.52	large
event	38	3.24	4.13	0.89	70	0.004	Yes	−0.28	small
Computing Attitudes									
computing interest	38	6.29	6.26	−0.03	75	0.939	No	−0.04	negligible
PDC interest	38	5.76	5.63	−0.13	120	0.568	No	0.09	negligible
PDC importance	38	5.74	6.13	0.39	62	0.029	Yes	−0.24	small
computing understanding	38	4.50	5.47	0.97	27	0.000	Yes	−0.36	medium
PDC understanding	38	3.32	4.95	1.63	49.5	0.000	Yes	−0.60	large

Table 4.17: USI CS1 Approximate Timing of the Unplugged "More Processors Are Not Always the Best" Activity.

Activity	**Duration**
Introduction to PDC: motivation and initiative	5 min
What is parallel computing? conceptual overview (partitioning, allocation, execution, communication) plus two short videos	6 min
Group formation and activity setup: teams of 5–6, rotation of the "processor" role, and the rule that each participant works in their own local buffer	5 min
Phase 1 — Uniprocessor sequential: one participant writes the sentence, then indexes each character; a second participant times the run	6 min
Phase 2 — Uniprocessor multitasking: the same participant interleaves writing and indexing character by character; time compared against Phase 1	6 min
Phase 3 — Two-processor parallel: one participant writes while a second indexes simultaneously; a third times, taking the slower processor as the overall time	7 min
Phase 4 — Multiprocessor parallel: two participants write sentence halves and two index them, with the dependency between indexers introducing wait overhead; a fifth participant times the slowest processor	9 min
Phase 5 — Amdahl rebalance: tasks are redistributed and a new task added (per-word character counts) to illustrate the limits of adding processors versus adding work	9 min
Group discussion: what aspects of parallel computing did each team observe across the phases?	5 min
Whole-class debrief and questions: comparison of recorded times, speedup, load balancing, dependency overhead, and Amdahl's law	9 min
Post-activity student engagement survey (Google Form)	4 min
Closing and takeaways	4 min
Total	**75 min**

Phase	Resource Configuration	Task Breakdown & Execution Dynamics	Performance Metric Logic
1. Uniprocessor Sequential	1 Active Worker, 1 Timer	Participant 1 executes Task 1 completely, followed sequentially by Task 2 on a single sheet.	Establishes the core baseline execution time.
2. Uniprocessor Multi-Tasking	1 Active Worker, 1 Timer	Participant 1 interleaves the workload character-by-character (writes a letter, then immediately writes its index).	Benchmarks the operational impact of context-switching/interleaving against the serial baseline.
3. Two-Processor Parallel	2 Active Workers, 1 Timer	Participant 1 writes the complete sentence (Task 1) while Participant 2 concurrently indexes it (Task 2) on their respective sheets from given start/end boundaries.	Total time is bound by the slower of the two concurrent workers.
4. Multi-Processor Parallel with Communication Overhead	4 Active Workers, 1 Timer	Data Partitioning: Workers 1 and 2 split the string writing workload ("more processors are" vs. "not always the best"). Workers 3 and 4 handle indexing. Worker 4 must explicitly wait to receive the final index boundary from Worker 3 before continuing.	Illustrates idle processor time and synchronization overhead. Total time is bound by the slowest processor.

Figure 4.10: USI CS1 Progressive Operational Activity Phases to Benchmark Execution Time

4.7 Conclusion

This chapter demonstrates that PDC concepts can be introduced meaningfully in a CS1 course without displacing core object-oriented objectives or consuming additional instructional days. The evidence assembled at USI across the Fall 2024 and Fall 2025 offerings of CS 258 supports the claim. A small number of well-placed "unplugged" activities, paired with targeted code implementations or "plugged-in" activities, and asynchronous visual demonstrations, produced statistically reliable gains on the PDC concepts those activities were designed to teach.

- Lessons Learned
  - Pre- and post-test analyses revealed that certain PDC concepts failed to show expected learning gains because of specific activity design limitations. For instance, the lack of improvement regarding race conditions stems directly from how the tasks were structured. Both the Flag Maker and Penny Sorting activities involve distributing and decomposing independent workloads across processors without incorporating shared mutable state. Consequently, while students gained an excellent intuition for decomposition, speedup, and load balancing, they did not encounter or understand race conditions. This underscores a critical pedagogical lesson: concepts that are merely mentioned during an activity, rather than actively integrated, can be overshadowed and must be carefully reevaluated during future activity redesigns.
  - Both the Fall 2024 and Fall 2025 cohorts suffered significant data attrition due to inconsistent, missing student identifiers. In Fall 2024, the sample size dropped from 27 pre-course respondents to 19 post-course respondents, ultimately yielding only 13 uniquely matched pairs after filtering out duplicate submissions and malformed email addresses. Similarly, the Fall 2025 cohort shrank from 32 pre-

course to 27 post-course respondents, resulting in just 23 successfully matched pairs. To prevent this data loss in future iterations, surveys will be tied to a small grade incentive to boost completion rates, and by using standardized coded student IDs to stay anonymous and consistent.

- Teaching Tips for Instructors
  - Before adopting an activity, ask which PDC concept its physical mechanics actually instantiate. For example, Flag Maker instantiates decomposition, speedup, and load imbalance, and it teaches those three superbly.
  - For effective implementation, activities should use existing sessions rather than additional ones, and be positioned so that the accompanying code implementations are reinforced versions of existing assignments or projects.
  - Pre-populate the identifier field, or use a roster-linked code for collecting student data. This single change would preserve a substantial fraction of the data lost pre- and post- survey collection.
- Inferences and Limitations
  - the language of instruction changed from C# to Java between the two semesters. Since the design does not support a cross-semester comparison in any case, this is not a confound to be adjusted for a limit on generalization: the Fall 2025 result is a result about a Java-based CS1.
  - the samples are small and subject to attrition. Twenty-three matched pairs in Fall 2025 and 14 in Fall 2024 leave little power for subgroup analysis.
  - this chapter's data do not speak to whether CS1 learning outcomes were preserved. The claim that PDC content did not displace core objectives rests on the structural fact that no instructional days were added and no core topics were removed.

4.7.1 Future Work

Several extensions follow directly from the limitations above. The most immediate is instrument redesign for establishing a stable, roster-linked identifier, and addition of a grade for completing the surveys. Beyond the CS1 boundary, the durability of these gains remains open. Tracking the Fall 2025 cohort into CS 358 (CS2) and later coursework would establish whether early intuitions persist and whether they accelerate later learning.

Appendix

Github Link to Instructional Material

The USI CS1 repository containing relevant PDC intervention material as described in this chapter can be found at: *USI-CS1-PDC-Github*.

Detailed Pre and Post Syllabus

The Parallel and Distributed Computing (PDC) intervention was implemented in CS 258, Introduction to Object-Oriented Programming, the CS1 course at the University of Southern Indiana. CS 258 is a three-credit, Java-based course covering fundamental object-oriented programming using Gaddis's Starting Out with Java: Control Structures through Objects (Pearson Revel Ed.). The pre-intervention syllabus followed a conventional CS1 structure, listing nine course learning objectives centered on introductory OOP concepts, object-oriented principles, design patterns, debugging, multithreaded applications, group collaboration, code review and documentation, and a final project, with no explicit treatment of parallel and distributed computing. The post-intervention syllabus preserved this structure but added a dedicated learning objective on parallel computing fundamentals, under which students explain sequential versus parallel computing, data parallelism, speedup, and parallel overhead, and apply existing parallel methods to optimize sequential code. The accompanying course event calendar integrates the PDC interventions directly into the existing nine module schedule (FlagMaker exercise within Module 4 (Loops and Files), a Penny Sorting activity as the Module 9 final project, and a Greenfoot event-driven programming demonstration). The pre- and post-course experience surveys were collected in the first and last weeks of the semester respectively. In this way, PDC concepts were woven into an otherwise standard CS1 sequence without displacing existing core course content.

Organization of Material

The `chapter4-USI` directory is organized as follows:

```
chapter4-USI/
  Detailed_Pre_and_Post_Syllabus/
  Evaluation_Data_and_Analysis/
  Plugged-In_Activities/
  Unplugged_Activities/
  USI-CS1_CourseCalendar_Fall2025.pdf
  readme.md
```

Detailed_Pre_and_Post_Syllabus

Purpose: This directory (available at *Detailed-Pre-Post-Syllabus*) contains detailed syllabus materials for pre- and post-course assessments.

Contents: Documentation and resources related to course prerequisites and learning outcomes established before and after course instruction.

Usage: Reference materials for understanding the syllabus structure and course requirements.

Evaluation_Data_and_Analysis

Purpose: This directory (available at *Evaluation-Data-and-Analysis*) houses data collection and analytical materials used for course evaluation.

Contents: Assessment data, evaluation metrics, and analysis documentation for measuring student learning outcomes and course effectiveness.

Usage: Source for course evaluation reports, student performance metrics, and instructional efficacy analysis.

Plugged-In_Activities

Purpose: This directory (available at *Plugged-In-Activities*) contains computer-based instructional activities that require digital resources and technology.

Contents: Programming exercises, computational thinking labs, hands-on coding activities, and digital learning modules designed to develop technical competencies.

Usage: Classroom instruction materials for technology-integrated lessons and practical programming exercises.

Unplugged_Activities

Purpose: This directory (available at Unplugged-Activities) contains non-digital instructional activities focused on foundational concepts.

Contents: Paper-based exercises, kinesthetic learning activities, collaborative group work, and conceptual activities that develop problem-solving skills without direct technology use.

Usage: Complementary classroom materials for reinforcing core concepts and engaging diverse learning styles.

USI-CS1_CourseCalendar_Fall2025.pdf

Purpose: Official course calendar (available at *USI-CS1-Course-Calendar-Fall2025*) for the Fall 2025 CS1 course at the University of Southern Indiana.

Contents: Academic schedule including lecture dates, assignment deadlines, examination dates, and important institutional dates.

File Type: PDF document

Usage: Primary scheduling reference for students and instructors.

readme.md

Purpose: Quick reference documentation for the USI module.

Contents: Overview, navigation guide, and key information about accessing and utilizing resources in this directory.

Usage: Entry point for users to understand the module's organization and purpose.

Survey and Other Anonymized Data and Analysis

On the GitHub repository, the directory *Evaluation-Data-and-Analysis*, contains the de-identified and anonymized data and the analytical materials used for the evaluation of the PDC interventions.

Chapter 5

CS1 Course Package – University of Nebraska–Lincoln

Scalable Codeless and Low-Code PDC Modules for Large and Online CS1 Contexts[†]

Chris Bourke[1]

[1]University of Nebraska–Lincoln

chris.bourke@unl.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 165
5.1 Introduction . . . 167
5.2 How CS1 was changed and PDC topics integrated . . . 167
5.2.1 Integrated Syllabi (Pre and Post) . . . 168
5.2.2 Challenges . . . 168
5.2.3 Student Time Commitment . . . 168
5.3 New Unplugged Activities . . . 169
5.4 New Plugged-in Activities . . . 169
5.4.1 Codeless PDC Modules . . . 169
5.4.1.1 Quick Start . . . 169
5.4.2 Structure . . . 169
5.4.2.1 Module 1.0 - Asynchronous & Parallel Computing . . . 170
5.4.2.2 Module 2.0 - Parallel Computing Speedup . . . 171
5.4.2.3 Module 3.0 - Distributed Computing . . . 172
5.5 Course-wide Pre and Post Course Surveys . . . 172
5.6 Other Activities and Ideas . . . 175
5.6.1 Lab: Processing a Remote Data Source . . . 175
5.6.2 Programming: Vibe Coding a Client-Server Battleship Game . . . 175
5.7 Conclusion . . . 175
Appendix . . . 177

Chapter figures

5.1 A screenshot of Module 1.0's Parallel Computing Demo . . . 171
5.2 Example run of parallel password cracking. The contents of a dictionary are split among a specified number of threads ("down" to "mallorca" for example). . . . 172
5.3 A screenshot of Module 3.0's Producer-Consumer Demo . . . 173

Chapter tables

5.1 UNL CS1 Learning Outcomes & Blooms Levels . . . 165
5.2 Week-to-Week Modules for UNL's CS1 . . . 168
5.3 Time Commitment for CS1 Activities - UNL . . . 169
5.4 Survey Results for CS1 - UNL. . . . 174

Abstract

At the University of Nebraska–Lincoln, Parallel and Distributed Computing (PDC) concepts were introduced into multiple sections of a Computer Science I (CS1) course using "codeless" modules consisting of visualizations, simulations, and demonstrations.

These modules are codeless because they do not require students to write or understand code. Instead, students read a short introduction to a PDC concept and then engage with a web-based visualization and/or (code-based) demonstration reinforcing the concept. The codeless nature of these modules makes them suitable for computing and non-computing majors and can be incorporated into a traditional course or assigned asynchronously for online and hybrid courses.

In addition, a lab that included distributed computing was updated and modernized while a new distributed programming assignment was created.

Descriptor Elements

Programming Language Employed: C, Java

Relevant core courses: CS1

Pre-requisites: None

Relevant PDC topics: Table 5.1 shows the activity-level learning outcomes added to UNL CS1 course with the incorporation of Parallel & Distributed Computing topics. Blooms levels are listed for each module and activity (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV), and Creating (CR)).

Table 5.1: UNL CS1 Learning Outcomes & Blooms Levels

PDC Concept	**Module Content & Blooms Level**		
	Codeless Modules 5.4.1	USGS Data Lab 5.6.1	Distributed Battleship Assignment 5.6.2
PDC Vocabulary/Foundations	Remembering (RE)	Remembering (RE)	Remembering (RE)
Data Parallelism	Understanding (UN)		
Speedup and Benchmarking	Understanding (UN)		
Distributed Computing	Understanding (UN)	Understanding (UN)	Applying (AP)
Event-Driven Programming	Understanding (UN)		

Learning outcomes: Students should have exposure to parallel, distributed, and asynchronous models of computing and appreciate the potential speed-up these models provide.

Context for use: The University of Nebraska–Lincoln is a large Midwestern R1 institution. The School of Computing is housed in the College of Engineering and offers multiple computing majors (CS, CE, SE, DS, Robotics) and minors and has nearly 1,000 undergraduate computing students. The sections of CS1 in which these modules were deployed include CSCE 155E which uses C as its programming language and mostly consists of CE, CS, and EE majors with an enrollment of about 100; and CSCE 155H an honors section that covers both C and Java in parallel and consists mainly of CS and CE majors who have demonstrated aptitude. Both sections are primarily new freshman students.

These courses are offered in a hybrid manner with scheduled lectures both in-person and livestreamed with recordings available immediately after. Weekly lab and hack sessions are in-person but student may complete their assignments outside of these lab sessions.

5.1 Introduction

Today, large-scale data centers and AI-driven applications are ubiquitous. Underlying these computing paradigms is high-performance computing (HPC), broadly encompassing parallel and distributed computing (PDC). However, many computing graduates do not possess the skills necessary to work effectively in HPC environments. One contributing factor is that computing curricula—particularly at the introductory level—have not kept pace with the realities of modern computing. The last widely adopted single-core processor architectures were introduced well over a decade ago, yet introductory courses are still often taught as if computation occurs in a single-core, single-machine context. Consequently, PDC topics are typically confined to upper-level courses or offered as *optional* electives, which are among the least frequently taken [7].

There is a clear opportunity to introduce PDC concepts *earlier* in the curriculum (e.g., CS1/CS2) and to extend this exposure to non-computing majors. Doing so can improve both understanding and appreciation of PDC while enhancing graduates' readiness for the high-performance computing workforce. To address this need, we have developed a series of "codeless" instructional modules designed to introduce foundational PDC concepts to introductory computing students. Our approach is *codeless* in that it does not require students to write—or even initially understand—code. Instead, the modules consist of web-based interactive visualizations and simulations, making them accessible regardless of programming experience or technical background.

This work builds on a substantial body of research demonstrating that interactive visualizations can significantly enhance student learning outcomes [53, 56, 64]. Our approach also addresses common scalability limitations: the modules are designed to be flexible, enabling integration into a wide range of courses and programming environments, or deployment as standalone, asynchronous learning resources.

In addition, we also detail two additional activities (a lab and a programming assignment) that give students exposure to distributed computing models (cf. Section 5.6).

We did not adopt any unplugged activities as this is a large course that is offered in an asynchronous manner. Our codeless modules are a useful alternative for instructors who cannot easily run synchronous, in-class activities due to time, resources, or scale. Instead, these modules can either be used in class as complementary demonstrations to other PDC material or assigned asynchronously.

The rest of this Chapter is organized as follows. An overview of the changes is presented in Section 5.2. The codeless modules are detailed in Section 5.4.1 while the USGS Data lab is discussed in Section 5.6.1 and the Battleship assignment is in Section 5.6.2.

5.2 How CS1 was changed and PDC topics integrated

The codeless PDC modules were first introduced into two CS1 sections in fall 2024 while the lab and programming assignment were introduced in subsequent years. The codeless modules were assigned to be completed outside of class asynchronously over the period of 8 weeks so that no structural or curricular changes were necessary, making integration easy.

Table 5.2: Week-to-Week Modules for UNL's CS1

Module	Topics	PDC Notes
1	Course Introduction, Git	
2	Basics: variables, operators	Pre-test data collection
3	Conditionals	PDC Modules assigned (parallel, asynchronous computing), see Section 5.4.1
4	Loops	
5	Functions, Unit Testing	
6	Pointers, Pass-by-reference, error handling	
7	Arrays	
8	Debugging, GDB	
9	Strings	
10	File I/O	
11	Encapsulation, structures	USGS Data Lab (distributed computing), see Section 5.6.1
12	Recursion	
13	Searching & Sorting	PDC Modules Due
14	Artificial Intelligence	Post-test data collection, Battleship assignment (distributed computing), see Section 5.6.2

The lab was an update of an existing lab while the programming assignment was assigned during the final week of class which did not previously have an assignment.

5.2.1 Integrated Syllabi (Pre and Post)

The syllabus for this course is available on GitHub. The following learning outcome was added as a result of this project: "Students will have exposure to parallel and distributed computing." No other changes were made. Table 5.2 contains a week-by-week topic list which are organized into modules.

5.2.2 Challenges

The primary challenge was in the development of the modules and material which required a substantial amount of time and effort. However, it was viewed as an investment because the modules, lab, and assignment would be reusable over several years, requiring only minimal maintenance and occasional updates.

If modules are assigned asynchronously, it is recommended that regular reminders are sent out to students on a regular basis with increased frequency closer to the due dates.

5.2.3 Student Time Commitment

Students were given 8 weeks to complete the modules which were estimated to take about 3-4 hours in total. The lab is designed to be completed in 1.25 hours and is typically assigned

Table 5.3: Time Commitment for CS1 Activities - UNL

Activity	Typical Student Time
Module 1.0	2.0 hours
Module 2.0	0.5 hours
Module 3.0	1.0 hours
Encapsulation Lab (USGS)	1.25 hours
Vibe Coding (Battleship)	2 - 5 hours

to be due by the end of the day to accommodate students who do not complete the lab within the lab period. Finally, the assignment had a large variation of completion times because it was more open ended. Some students completed it extremely quickly if the GenAI was able to generate a complete solution. Others took longer if the GenAI was not as efficient or produced buggy code. Yet others took even longer because they wanted to improve their game and add additional features that were not required. This is summarized in Table 5.3.

5.3 New Unplugged Activities

No unplugged activities were developed or adopted for this course. Enrollment is typically large and the hybrid nature of the course does not readily lend itself to in-class unplugged activities. This chapter instead contributes a scalable codeless/asynchronous pathway.

5.4 New Plugged-in Activities

5.4.1 Codeless PDC Modules

5.4.1.1 Quick Start

The codeless PDC modules can be adopted or adapted into any CS1 course as they do not require any software installation nor programming background. Instructors can assign the modules or use the visualizations as part of their lectures or presentations as they are hosted at `https://go.unl.edu/PDCatUNL`. Each module should take no more than 2 hours to complete including reading. They can be assigned as in-class activities or asynchronously. Instructors wishing to adapt or extend the modules can find the code hosted at `https://github.com/cbourke/PDCatUNL`.

5.4.2 Structure

Each codeless module is structured as follows: students are asked to read a short introductory text explaining the concepts which will be demonstrated through simple browser-based activities. They then are asked to engage in interactive visualizations or simulations which do not require them to develop or understand any code. Some of the modules include an additional demonstration which requires the student to compile and run a program but does

not require them to write, edit, or read any code. Finally, students are asked to reflect on the concepts by answering a series of questions.

The reading material has been written at a level that does not require a deep understanding of PDC concepts. The material deliberately omits many details that would be necessary to gain expertise, and instead favors describing concepts in a more abstract manner which should make them more accessible to a wider audience. For example, we do not distinguish between data-parallel and task-parallel models, nor do we provide rigorous definitions of cores, threads, or processes. However, we do mention these concepts in a way that early computing students should be able to relate to and understand.

The simulations include both web-based visualizations and pre-written, code-based simulations. The web-based visualizations are interactive applications inspired by the highly successful utilization of similar technologies introduced to visualize programs [64], data structures and algorithms [56], and automata [53]. They can be run on any web browser without special environments or software installations. The code-based simulations include both C and Java versions to accommodate students in different introductory courses. However, we still describe them as "codeless" because students are not required to write or design any code. They are required to compile and run the programs, but we include complete, minimal instructions similar to what they should have already seen in other classes with programming assignments. It is unavoidable to require a proper C or Java environment for compiling and running these simulations, but we have tried to minimize external dependencies as much as possible.

At the end of each module, students are prompted to reflect on the reading and their interaction with the browser-based simulation and demonstration by answering a series of questions designed to ensure that: 1) they ran a sufficient number of use cases in each simulation; and 2) their simulation experience directly traces back to the reading which is intended to reinforce understanding. For example, they are asked how much time a simulation took to execute with different numbers of threads and are then asked to deduce specific *reasons* for trends they saw (*i.e.*, seeing a roughly double speed-up when doubling the threads, but not seeing an improvement beyond the number of cores available on the system). To help understanding and avoid misconceptions, suggested answers are provided based on our own simulations. This reflects our intent for the modules to be a formative, rather than summative, assessment.

5.4.2.1 Module 1.0 - Asynchronous & Parallel Computing

The first module introduces students to asynchronous and parallel computing through several web-based visualizations which contrast with sequential and synchronous computing. The consequences of each type of computing are highlighted in the visualizations and demonstrations. In one example, students are given a browser-based GUI which executes a synchronous, blocking action that freezes the UI, and an asynchronous action which does not.

The main demonstration is a parallel computing simulation and visualization in which the user can specify a number of generic computing tasks, a number of processors to execute them, and how long each task will take (*e.g.*, 2-6 seconds). The simulation then queues the tasks for each processor to execute in order. The simulation reports the status of each task

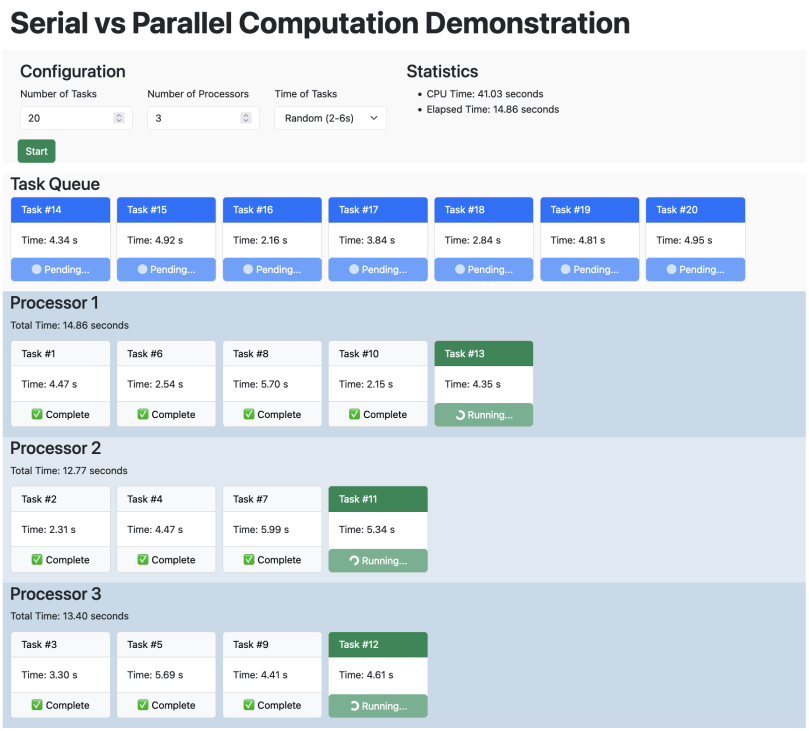


Figure 5.1: A screenshot of Module 1.0's Parallel Computing Demo

as well as total wall-clock time compared to processor time. Figure 5.1 contains an example screenshot of the visualization.

5.4.2.2 Module 2.0 - Parallel Computing Speedup

The second module introduces students to the concept of threads. It is based on a brute-force, dictionary-based, password-cracking scheme which uses an "embarrassingly parallel" approach. We give students a basic explanation how password hashing works to motivate the activity as well as provide early exposure to security issues such as the importance of strong passwords. They are provided both sequential and parallel password-cracking programs with a C version using `pthreads` and a Java version using `Callables`. We include full instructions on cloning, compiling, and running the programs, as well as how to read and interpret the results. No actual coding is necessary to run and observe the simulations.

The demonstration exposes students to the potential performance improvements of parallel computation and its limits. They observe (roughly) proportional speed ups until the number of threads exceeds the number of processors after which no further improvement is observed. When they run the program, they can specify how many threads to create. They are instructed to run the program multiple times with 1, 2, 4, *etc.* threads and observe the trends. An example run of the C version's output can be found in Figure 5.2.

We note that the demonstration is data-parallel because the dictionary listings are split evenly among each thread, so we chose password hashes which ensures the program does not get "lucky" by finding the hash early in any particular data slice, regardless of how many threads are specified.

```
creating 4 threads...DONE
Starting threads...
Starting thread 1...
Starting thread 2...
Thread 2 hashing [61851, 123702) = [down, mallorca]...
Starting thread 3...
Thread 1 hashing [0, 61851) = [a, downscaled]...
Starting thread 4...
Thread 3 hashing [123702, 185553) = [mallorcan, rotc]...
Thread 4 hashing [185553, 247406) = [rotch, zzz]...
Thread 1 did not crack password
Thread 4 cracked password: 0x618e0fcd...b0b => zzz42
Signaling other threads to terminate...
All threads DONE
Time:
218.541u 0.015s 0:54.82 398.6% 0+0k 4888+0io 0pf+0w
```

Figure 5.2: Example run of parallel password cracking. The contents of a dictionary are split among a specified number of threads ("down" to "mallorca" for example).

5.4.2.3 Module 3.0 - Distributed Computing

The final module introduces students to a producer-consumer pattern often used in distributed computing environments. Students are provided another browser-based visualization as well as a C and Java implementation to investigate. In all versions, generic tasks represent requests or work which are simulated through HTTPS requests to a web-service that causes a delay before responding. The length of the delay can be specified or randomized.

All of the simulations allow students to specify the number of producers and of consumers. Once started, each producer will generate tasks at random intervals and place them into a task queue for processing. When a consumer is available, it takes the next available task from the queue and executes it. The simulation will continue until stopped by the user. An example screenshot of the web-based visualization can be found in Figure 5.3.

As with the other modules, the questions point students to observe several scenarios, such as having equal or unbalanced numbers of producers or consumers. Students are then asked to predict the consequences of those different scenarios.

5.5 Course-wide Pre and Post Course Surveys

To determine if our codeless modules were effective at teaching basic PDC concepts, we administered pre and post exams and surveys.

To measure students' knowledge and understanding of PDC concepts, they were asked to provide free-form text answers to the following questions. For each of the four identified PDC concepts: sequential (S), asynchronous (A), parallel (P), distributed (D) computing,

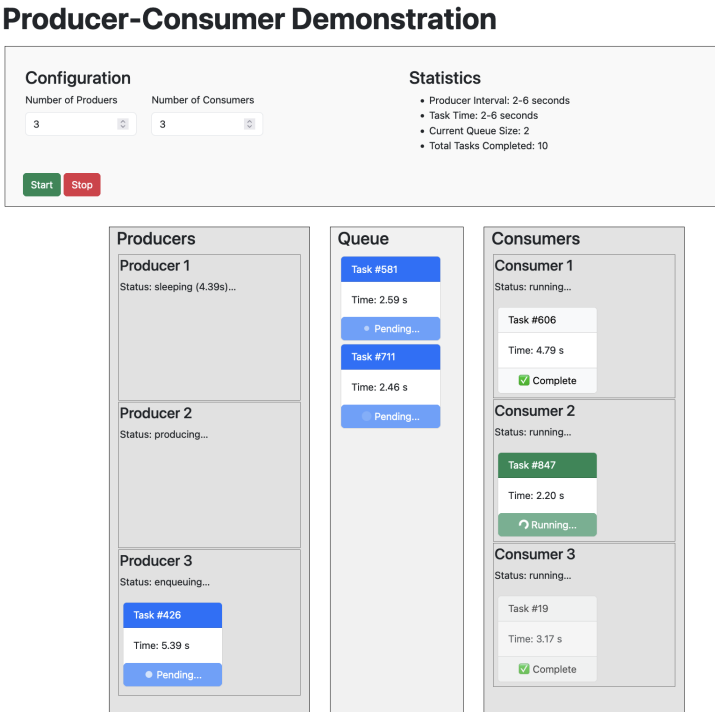


Figure 5.3: A screenshot of Module 3.0's Producer-Consumer Demo

students were asked to provide:

1) a definition (D) of the concept; and
2) an example (E) of the concept

To measure attitudes, students were first asked to agree or disagree with the following statements:

A1 In general, my interest in computing is high
A2 My interest in learning about parallel and distributed computing is high
A3 In general, I believe learning about parallel and distributed computing is important
A4 I believe I have a good understanding of computing
A5 I believe I have a good understanding of parallel and distributed computing

Student answers were collected using a 7-point Likert scale (7 = strongly agree, ... 4 = neutral, ... 1 = strongly disagree).

Results were very strong. Across all dimensions, our codeless PDC modules showed statistically significant and substantial (with respect to effect size) improvement in understanding and appreciation of PDC. For a complete analysis see [6] and [5].

In addition, we administered the common pre/post survey (see Section 1.5); the results can be found in Table 5.4. Computing background data indicated a negligible to small improvement which reflected the already-high ratings students self-reported. This is expected as this population are computing majors and would already have a substantial computing background. The programming language familiarity was as expected. The main source of data was from courses that used C as the introductory language and so it showed a large improvement in familiarity while other languages did not. Likewise, the computing

concepts showed medium improvements overall. Some of the concepts only showed a small improvement but this is expected as classes and objects are not directly covered when using C (we cover structures and encapsulation but do not use this terminology). Finally, the results of the PDC concepts were consistent with the observations published in [6] and [5]. Students showed substantial improvements in familiarity with respect to parallel and distributed computing.

Table 5.4: Survey Results for CS1 - UNL. Sample size was 125 and included data from fall 2024 and fall 2025 semesters. The pre- and post means are reported along with the difference (Diff). Statistical significance is measured using a non-parametric Wilcoxon signed-rank (paired) test. The Wilcoxon statistic (W) is reported along with the p-value. Significance is indicated as Yes using $p < 0.05$. Effect size is reported using Cliff's Delta (δ); a negative value indicates post-survey increase. The interpretation of the effect size is reported (negligible, small, medium, large) as suggested by [55].

Item	Pre	Post	Diff	W	p-value	Sig.	Cliff's δ	Interp.
Computing Background								
Computer Use	5.80	5.84	0.04	884.5	0.643	No	-0.04	negligible
Programming	3.61	4.14	0.53	567.5	0.000	Yes	-0.21	small
Programming Compared to Peers	3.53	3.97	0.44	866.0	0.000	Yes	-0.17	small
Programming Languages								
C	2.50	4.19	1.70	164.5	0.000	Yes	-0.68	large
C++	1.54	1.97	0.42	286.0	0.000	Yes	-0.21	small
C#	1.42	1.66	0.24	104.0	0.001	Yes	-0.11	negligible
Java	2.70	2.84	0.14	327.0	0.101	No	-0.02	negligible
JavaScript	2.26	2.26	0.00	466.0	0.928	No	-0.01	negligible
PHP	1.13	1.23	0.10	8.0	0.022	Yes	-0.04	negligible
Python	2.74	2.90	0.16	610.0	0.074	No	-0.07	negligible
Scratch	2.06	2.18	0.12	341.5	0.158	No	-0.07	negligible
Visual Basic	1.15	1.20	0.05	47.0	0.264	No	-0.04	negligible
other	1.82	1.85	0.02	540.0	0.995	No	-0.02	negligible
Computing Concepts								
Data/Variable Types	4.34	5.40	1.06	160.0	0.000	Yes	-0.33	medium
Arithmetic Expressions and Operators	4.35	5.22	0.86	266.5	0.000	Yes	-0.25	small
Branching	4.11	5.21	1.10	230.5	0.000	Yes	-0.32	small
Loops	3.97	5.21	1.24	186.0	0.000	Yes	-0.35	medium
Calling functions/methods	3.76	4.99	1.23	273.0	0.000	Yes	-0.35	medium
Arrays	3.13	4.61	1.48	375.0	0.000	Yes	-0.44	medium
Writing static functions/methods	2.95	4.34	1.38	354.0	0.000	Yes	-0.43	medium
Writing functions/methods	3.22	4.70	1.48	224.5	0.000	Yes	-0.44	medium
Writing classes	2.70	3.62	0.92	433.5	0.000	Yes	-0.32	small
Creating objects	2.62	3.48	0.86	422.0	0.000	Yes	-0.31	small
PDC Concepts								
Parallel Computing	1.66	2.70	1.05	526.5	0.000	Yes	-0.55	large
Distributed Computing	2.69	3.07	0.38	983.5	0.007	Yes	-0.17	small
Event Handling	1.76	2.68	0.92	483.5	0.000	Yes	-0.44	medium

5.6 Other Activities and Ideas

In addition to our codeless PDC modules, we introduced both a lab and a programming activity to give students exposure to distributed computing.

5.6.1 Lab: Processing a Remote Data Source

A lab on encapsulation (designing and implementing a C structure) had been a staple of this course for over 10 years. Previously, it involved connecting to various RSS feeds and displaying the most recent articles. However, in recent years RSS's popularity has declined. This gave us motivation to update this lab to instead process a live JSON data feed based on the Earthquake Tracker Remote Data Retrieval Assignment (see Section 1.4.3).

Students are not expected to write code to connect to the server and process the JSON responses directly. For our version, starter code is provided to do so and students then design and implement a structure to model the earthquake data. Students are also required to process the data to report the closest earthquake, strongest earthquake, etc. The lab is publicly available on GitHub and can easily be adopted or adapted for any CS1 course (using C).

5.6.2 Programming: Vibe Coding a Client-Server Battleship Game

As part of a larger effort to introduce AI and vibe coding into CS1, we developed a vibe coding activity that had students develop a client program to connect to a remote server to play a game of battleship. In this version, the client makes "moves" by submitting JSON requests to a remote server (calling "shots" on a 10×10 grid). The server responds with JSON data indicating a hit or miss and an updated score (each game has a limited number of shots) and game state (playing, win, loss, etc.).

The activity introduces students to distributed computing and network communication and requires formatting JSON data, performing HTTP requests and processing JSON responses in C. This would generally be considered to be too advanced even at the end of a CS1 course, but with the aid of an LLM, it is feasible. Students are introduced how to use an LLM, common pitfalls, and how to prompt the LLM, test the code it produces and then follow up to refine and debug its code.

In our first offering of this activity, over 75% of the students created an interactive client that was able to successfully communicate with the server and win a game of battleship. The exercise is publicly available on GitHub, however the Battleship server code is not publicly accessible. Individuals wishing to adopt this activity will need to develop and host their own server.

5.7 Conclusion

While most PDC efforts have focused on programming strategies for upper-level and elective courses, we believe there is value and opportunity in introducing these concepts much earlier in the computing curriculum. Our "codeless" approach is a simple, accessible,

and scalable way to introduce PDC concepts to both computing majors and non-majors as early as CS0. Our studies have demonstrated that our approach has statistically significant and substantial gains in both learning and appreciation for PDC topics.

For instructors thinking about developing similar content, advanced planning is highly advised. The codeless PDC modules are perfect for asynchronous delivery and take minimal class time. Developing the content required a substantial upfront investment in planning and time. However, once the modules were created, minimal maintenance was required.

Instructors have reported that it is easy to adopt the codeless modules as in-class demonstrations to complement their own PDC content coverage. The modular nature means that material can be adopted in part or in whole or remixed with other content. Using our modules in this manner does not require any additional development time and so this path would be ideal for instructors.

If instructors assign modules outside of class, you can expect students to complete all the modules within 3-4 hours total (less if only parts are assigned). One challenge may be to ensure that students complete the modules asynchronously. This can be addressed by adopting them as part of in-class activities or by creating small assessments to complement the modules.

Appendix

Github Link to Instructional Material

All general course material is hosted on GitHub. Additional material can be found at the CDER GitHub repository for this chapter (here).

Codeless Modules

- Source code for the codeless modules is hosted at the following GitHub repository for those wishing to extend or adapt the modules.
  `https://github.com/cbourke/PDCatUNL`
- A live version of the codeless modules is hosted at the URL below for those that want to adopt the modules directly. These can be used in class demonstrations or assigned to students asynchronously.
  `https://go.unl.edu/PDCatUNL`

Labs

- USGS Earthquake Data Processing Lab (distributed computing). The handout (see `readme.md` in the repo) and starter code are available for students to clone from the following GitHub repository.
  `https://github.com/cbourke/CSCE155-C-Lab11`
- Battleship (distributed computing). The handout is available in the CDER GitHub repository for this chapter under `Plugged_Activities/battleship.md`. Adopters will have to develop and host their own server.

Detailed Pre and Post Syllabus

See Section 5.2.1

Organization of Material

- The Codeless repository is a webapp Eclipse project primarily written in JavaScript with a Java backend. It also contains a `c` folder for all C code. Adopters should start with the live demo.
- The USGS Earthquake lab is a C project that is framework agnostic and can be directly cloned. The setup requires `apt-get` or similar package manager for dependencies. Adopters should start by reading the `readme.md` file at the repository root.

Survey and Other Anonymized Data and Analysis

No data is shareable as per our Institutional Review Board approved study. For a full aggregate analysis of student attitudes from the standard pre and post surveys, see Section 5.5.

Publications

Additional analysis can be found in the original publications [6] and [5] with electronic copies available here and here respectively.

Chapter 6

CS1 Course Package – Webster University

Visualization-First C++ Infusion through Animations, Simulations, and Game-Based Activities[†]

Xiaoyuan Suo[1] and Peter Maher[2]

[1]Webster University, xiaoyuansuo51@webster.edu
[2]Webster University, maherp@webster.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 181
6.1 Introduction . . . 182
6.2 Changes to CS1 and Integration of PDC Topics . . . 182
6.2.1 Original and Revised Syllabi . . . 183
6.2.2 Highlights . . . 183
6.2.2.1 Unplugged Activities . . . 183
6.2.2.2 Programming Demonstrations . . . 184
6.2.3 Challenges . . . 186
6.3 Conceptual Activities and Visualizations . . . 186
6.3.1 Animations and Visualizations . . . 186
6.4 New Programming Activities . . . 187
6.4.1 Zombie Attack Assignment . . . 187
6.5 Course-Wide Pre- and Post-Course Surveys . . . 188
6.6 Other Activities and Ideas . . . 189
6.7 Conclusion . . . 190
Appendix . . . 191

Chapter figures

6.1 Flag-coloring animation. . . . 187
6.2 Zombie Attack homework: red cells represent zombies and blue cells represent unaffected humans. . . . 188
6.3 Pre/post comparison of student self-reported experience with PDC-related concepts. Mean ratings are reported on a seven-point scale. Responses analyzed as unmatched groups; pre $n = 8$, post $n = 8$. . . . 193
6.4 Pre/post comparison of student self-reported programming experience across core C++ topics. Responses analyzed as unmatched groups; pre item $n = 8$, post item $n = 7$. . . . 194
6.5 Pre/post comparison of student attitudes, interest, and self-reported understanding. Responses analyzed as unmatched groups; pre $n = 8$, post $n = 8$. . . . 195

Chapter tables

6.1 PDC topics and their corresponding levels in the revised Bloom's Taxonomy. . . . 181
6.2 Sequential and parallel versions of summing values in an array. . . . 184
6.3 Summary of PDC activities integrated into the CS1 course. . . . 189
6.4 High-level syllabus with PDC enhancements in CS1. . . . 191
6.5 Descriptive pre/post differences in PDC-related experience. Percent favorable is the percentage of responses coded 5 or higher. Item-level $n = 8$. . . . 194
6.6 Descriptive pre/post differences in programming experience across core C++ concepts. Pre item $n = 8$; post item $n = 7$. The post functions topic summarizes the function-calls and writing-functions items. . . . 195
6.7 Descriptive pre/post differences in interest, perceived importance, and self-reported understanding. Item-level $n = 8$. . . . 195
6.8 Post-survey ratings of course learning activities. Item-level $n = 8$. . . . 196
6.9 Post-survey agreement with self-reported course-outcome statements. Item-level $n = 8$. 196

Abstract

This chapter presents a course transformation in an introductory computer science sequence (CS1) at Webster University, which is a primarily undergraduate institution, where Parallel and Distributed Computing (PDC) concepts are integrated without requiring prior background in systems or advanced mathematics. The target audience includes STEM undergraduates with little to no programming experience. The primary learning outcomes are for students to (1) understand core PDC ideas such as parallelism, speedup, and efficiency, (2) reason about workload distribution and performance trade-offs, and (3) connect conceptual understanding to simple implementations in C++. The transformation emphasizes accessible, low-barrier entry points through a combination of brief interactive visualizations, selective unplugged activities, and lightweight programming exercises, enabling students to grasp foundational performance concepts early in the curriculum.

Descriptor Elements

We designed a sequence of activities that progressively build students' understanding of PDC concepts, moving from intuitive, unplugged experiences to guided visualizations and finally to lightweight programming tasks.

Programming Language Employed: C++

Relevant core courses: CS1

Prerequisites: No prerequisites.

Textbook Starting Out with C++ from Control Structures to Objects, 10th edition, Tony Gaddis [23]

Relevant PDC topics and Bloom's taxonomy: Classification of PDC topics according to the revised Bloom's Taxonomy [2]: Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR).

Table 6.1: PDC topics and their corresponding levels in the revised Bloom's Taxonomy.

PDC Topic	**Bloom's Level**
Introduction to Parallel Computing	RE
Task Decomposition and Partitioning	UN
Speedup and Efficiency	AN
Performance Measurement and Benchmarking	AP

Learning outcomes:

1. understand core PDC ideas such as parallelism, speedup, and efficiency,
2. reason about workload distribution and performance trade-offs, and
3. connect conceptual understanding to simple implementations in C++.

Context for use: Webster University is a primarily undergraduate institution located in St. Louis. The Department of Computer and Information Sciences offers several undergraduate programs, including a B.S. in Computer Science with multiple emphasis areas. Introductory programming instruction is delivered through a two-course CS1 sequence consisting of COSC 1550 and COSC 1560. These courses introduce students to fundamental concepts in C++ programming, including variables, control structures, functions, arrays, pointers, file I/O, abstract data types (ADTs), basic object-oriented programming, and dynamic memory management. In addition to programming syntax and problem-solving skills, students develop foundational computational thinking skills through hands-on programming assignments and laboratory activities. Typical enrollment in each course ranges from approximately 12 to 25 students per semester.

6.1 Introduction

At Webster University, we have explored ways to introduce Parallel and Distributed Computing (PDC) concepts early in the curriculum, specifically within a CS1 course taught in C++. While PDC is traditionally reserved for upper-level courses, delaying exposure often reinforces the misconception that parallelism is an advanced or niche topic. In reality, parallel thinking, such as breaking problems into independent tasks, reasoning about concurrency, and understanding performance trade-offs, is a fundamental computational skill [63]. Introducing these ideas in CS1 helps students develop a more accurate mental model of modern computing, where multicore processors and parallel workloads are the norm rather than the exception.

Our approach introduces conceptual PDC activities around week 8 and integrates programming demonstrations and assignments in the later part of the 16-week course, after students have developed a solid foundation in loops, arrays, and functions. At this stage, students are ready to revisit familiar problems through a new lens: not just how to solve them, but how to solve them efficiently using parallelism. We use parallel search, an embarrassingly parallel summation problem, and an odd-even sort demonstration as accessible examples. These examples allow students to see immediate connections between sequential and parallel implementations without introducing unnecessary complexity.

6.2 Changes to CS1 and Integration of PDC Topics

We modified several traditional CS1 topics, particularly loops and arrays, to incorporate introductory parallel and distributed computing (PDC) concepts without significantly changing the overall structure of the course. Loops were used to introduce ideas such as data parallelism, independent task execution, and workload decomposition by demonstrating how repetitive operations can be divided among multiple workers. Arrays provided a natural context for discussing partitioning large datasets into smaller segments that can be processed concurrently. Simple examples such as searching, counting, and grid-based simulations were adapted to help students recognize opportunities for parallel execution while still reinforcing core programming concepts.

6.2.1 Original and Revised Syllabi

The overall structure of the CS1 sequence remained largely unchanged, allowing the integration of parallel and distributed computing (PDC) concepts without removing core introductory programming content. Instead, selected topics were enhanced with parallel thinking activities, demonstrations, and programming exercises. Traditional CS1 concepts such as loops, arrays, searching, and simulations were used as natural entry points for introducing decomposition, concurrency, workload partitioning, and performance considerations.

Table 6.4 (in the appendix) summarizes the major course topics and highlights where PDC-related modifications were incorporated.

6.2.2 Highlights

Core CS1 content remained largely unchanged, including variables, control structures, functions, arrays, pointers, object-oriented programming, and debugging. The primary changes involved embedding PDC concepts into existing topics through activities, simulations, and performance-oriented examples. Some highly language-specific or less essential STL-focused material was reduced or simplified to create instructional time for the new content.

6.2.2.1 Unplugged Activities

Before adopting animations, we relied on unplugged activities to introduce core PDC ideas without programming syntax [41]. One commonly used exercise was the *coin search* activity, where a group of students is asked to find a specific coin in a large jar. The task is first performed by a single student (sequential search), and then repeated with multiple students searching simultaneously (parallel search). This simple setup helps students experience, in a tangible way, how dividing work can reduce completion time, while also revealing limitations such as uneven workload distribution or coordination overhead.

We also used a *flag coloring* activity (Section 3.3.1) [58], where students are given a map or flag divided into regions and asked to color it under certain constraints (e.g., no adjacent regions sharing the same color). Students work either individually or in groups, with each person responsible for different regions, simulating task decomposition and concurrent execution. While the activity reinforces ideas such as independence of tasks and the need for coordination, many students perceived it as time-consuming. Based on this feedback, we began using short animations that convey the same concepts more efficiently, allowing us to move more quickly from intuition to implementation.

Timing and Teaching Tips The coin-search activity typically requires 15–20 minutes, while the full flag-coloring activity may require 30–50 minutes depending on class size and discussion. Before beginning, ask students to predict whether doubling the number of workers will halve the completion time. Afterward, discuss uneven workload distribution, coordination costs, and why measured speedup may differ from the ideal case. For shorter class periods, the corresponding animations can be used instead of the full unplugged activities. Animations can be found in Section 6.3.1.

6.2.2.2 Programming Demonstrations

After establishing these conceptual foundations, we introduce OpenMP as a practical tool for parallel programming in C++. However, this step presents its own challenges. Many students struggle with the syntax of compiler directives, particularly the meaning and role of `#pragma`. Rather than treating it as a purely syntactic feature, we frame `#pragma omp` as an instruction to the compiler that enables parallel execution of code regions. By tying this back to the earlier animations and activities, students can interpret OpenMP not as "magic," but as a mechanism for expressing the parallel ideas they already understand conceptually.

In the programming demo shown in Table 6.2, students generated vectors of random integer values and, using our provided code, compared sequential and parallel implementations of the summation operation. Because each array element can be processed independently, the example provided a simple introduction to data parallelism and reduction operations while remaining accessible to CS1 students.

Table 6.2: Sequential and parallel versions of summing values in an array.

Sequential Version	**Parallel Version**

`int sum_serial(`
`  const vector<int>& data)`
`{`
`  int sum = 0;`
 	
`  for (int i = 0;`
`  i < data.size();`
`  ++i) {`
`  sum += data[i];`
`  }`
 	
`  return sum;`
`}`
	`int sum_parallel(`
	`  const vector<int>& data)`
	`{`
	`  int sum = 0;`
	
	`#pragma omp parallel for reduction`
	`  (+ : sum)`
	`  for (int i = 0;`
	`  i < data.size();`
	`  ++i) {`
	`  sum += data[i];`
	`  }`
	
	`  return sum;`
	`}`

Listing 6.1 shows a modified bubble-sort example using OpenMP. Instead of comparing every adjacent pair sequentially, the algorithm alternates between even- and odd-indexed pairs, enabling independent comparisons to be executed in parallel.

Listing 6.1: OpenMP parallel odd-even sort example

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>  // Compile with OpenMP support (e.g., -fopenmp).

// Parallel Odd-Even Sort implementation
void parallelOddEvenSort(std::vector<int>& arr) {
    int n = arr.size();
```

```
    // The algorithm requires up to n total phases to guarantee sorting
    for (int phase = 0; phase < n; ++phase) {

        // Phase 0, 2, 4... -> Even Phase (pairs: 0-1, 2-3, 4-5...)
        // Phase 1, 3, 5... -> Odd Phase  (pairs: 1-2, 3-4, 5-6...)
        if (phase % 2 == 0) {
            // Parallelize the even-indexed pairs loop
            #pragma omp parallel for shared(arr, n)
            for (int i = 0; i < n - 1; i += 2) {
                if (arr[i] > arr[i + 1]) {
                    std::swap(arr[i], arr[i + 1]);
                }
            }
        } else {
            // Parallelize the odd-indexed pairs loop
            #pragma omp parallel for shared(arr, n)
            for (int i = 1; i < n - 1; i += 2) {
                if (arr[i] > arr[i + 1]) {
                    std::swap(arr[i], arr[i + 1]);
                }
            }
        }
    }
}

int main() {
    // Sample unsorted array
    // This small input demonstrates correctness, not speedup.
    // Use a much larger input for performance experiments.
    std::vector<int> data = {34, 7, 23, 32, 5, 62, 78, 4, 0, 11, 99, 14};

    std::cout << "Original_Array:";
    for (int num : data) std::cout << num << ",";
    std::cout << "\n";

    // Run parallel sort
    parallelOddEvenSort(data);

    std::cout << "Sorted_Array:";
    for (int num : data) std::cout << num << ",";
    std::cout << "\n";

    return 0;
}
```

Timing and Teaching Tips The parallel-sum demonstration typically requires 20–25 minutes, and the odd-even sort demonstration requires approximately 25–30 minutes. Begin with the familiar sequential version and ask students to identify which operations are independent before showing the OpenMP code. Use a sufficiently large input when discussing performance, since small examples may run more slowly in parallel because thread-creation and synchronization overhead dominate the computation.

6.2.3 Challenges

One of the primary challenges in integrating PDC concepts into CS1 was the limited amount of available instructional time. Introductory programming courses are already densely packed with foundational topics such as variables, control structures, functions, arrays, pointers, object-oriented programming, and debugging. To incorporate PDC material without overwhelming students, we modernized existing topics rather than adding entirely new units. Concepts such as data decomposition, concurrency, and workload partitioning were embedded into familiar topics including loops, arrays, and searching. To create space for these additions, some highly language-specific or less commonly used STL-focused content was reduced.

Another pedagogical challenge involved balancing conceptual understanding with programming complexity. Since many CS1 students are novice programmers, introducing advanced ideas such as synchronization, scalability, and communication overhead too early can increase cognitive load. To address this issue, we relied heavily on unplugged activities and animations as visual demonstrations before introducing code-based examples. Activities such as coin-search exercises, zombie outbreak simulations, and search demonstrations helped students develop intuition about parallel execution in an accessible way.

Designing effective programming assignments also became increasingly difficult due to the widespread availability of generative AI tools. Traditional introductory programming assignments can often be solved directly using AI-generated code, making it harder to evaluate genuine student understanding. As a result, we tried to redesign assignments to emphasize experimentation and debugging, but the efforts still need to be further evaluated.

At the same time, because many students had not yet taken courses in computer architecture or operating systems, advanced performance concepts had to be presented conceptually rather than mathematically. The instructional approach therefore emphasized intuition-building and empirical experimentation using lightweight tools and small-scale examples that could run on standard student laptops without requiring specialized hardware.

6.3 Conceptual Activities and Visualizations

6.3.1 Animations and Visualizations

The animation links below point to GitHub source files rather than third-party preview services. Download each HTML file and open it in a web browser to run the animation.

A key challenge in teaching PDC at this level is abstraction. Concepts such as threads, synchronization, and compiler directives can be difficult for beginners to grasp, especially when introduced purely through code. To address this, we incorporate visual and interactive teaching strategies. First, we use lightweight animations (Parallel Search Animation Source) and (Parallel Flag Coloring Animation Source Figure 6.1) to illustrate how tasks are divided and executed concurrently. These animations help students build an intuitive understanding of parallel execution, such as how multiple "workers" operate simultaneously, how workload is distributed, and why certain problems scale better than others. By visualizing execution, students can more easily connect code behavior with underlying computational processes.

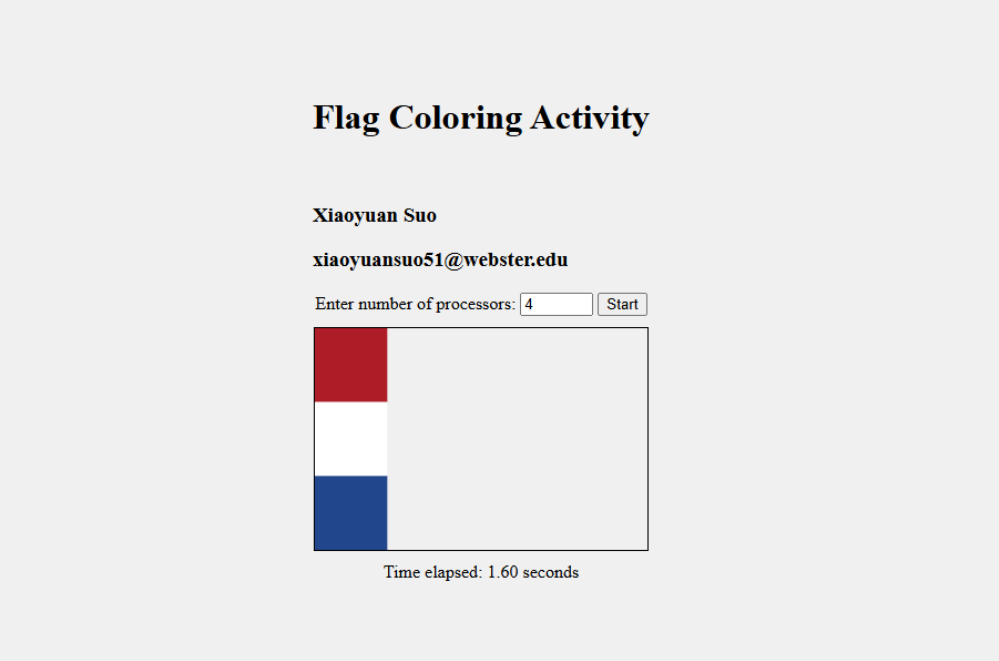


Figure 6.1: Flag-coloring animation.

Timing and Teaching Tips Each animation can be used in approximately 10–15 minutes. Ask students to predict the next step or identify which tasks can execute concurrently before running the animation. Pause at key points to discuss task decomposition, dependencies, synchronization, and workload balance. The animations are most effective when immediately connected to a familiar CS1 example, such as array searching, loops, or a simple OpenMP demonstration.

6.4 New Programming Activities

6.4.1 Zombie Attack Assignment

As for the assignment, students complete a *"zombie attack"* homework problem[9]. In this assignment, students model the spread of a zombie infection across a grid-based environment and investigate how the simulation can be decomposed and executed in parallel. The task is intentionally designed to be conceptually approachable while exposing students to fundamental PDC ideas, including data partitioning, locality, synchronization, and scalability. Students are encouraged to experiment with different decomposition strategies (e.g., dividing the grid into subregions) and to reason about communication overhead and performance trade-offs as the number of processing units increases. This assignment serves as a bridge between conceptual understanding and computational thinking, reinforcing key ideas in a creative and engaging context; see Figure 6.2.

To support this activity, I developed a lightweight animation that visually demonstrates the infection propagation process: (Zombie Attack Animation Source). To clarify, the animation illustrates the spread using a single zombie; the C++ assignment uses a larger

grid to support discussion of decomposition and performance. Instructors should check the assignment instructions and supporting code for the required initialization and update rules before adoption. The Python-CS2 variant is a separate resource discussed in Chapter 16.

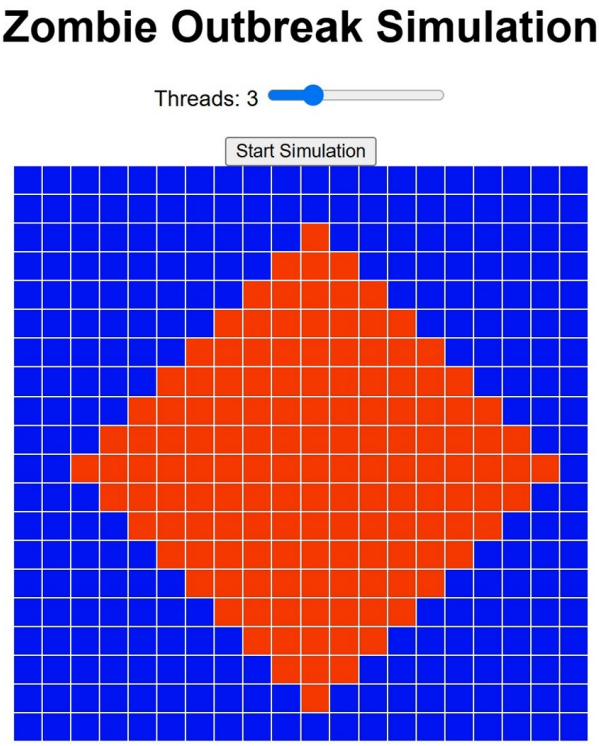


Figure 6.2: Zombie Attack homework: red cells represent zombies and blue cells represent unaffected humans.

Timing and Teaching Tips Introduce the animation in 10–15 minutes before assigning the programming task, which typically requires one to two weeks. Begin with a small grid and one zombie so students can verify each update manually, then increase the grid size and number of zombies. Emphasize correctness before performance, and ask students to identify boundary cells, shared data, and the points at which synchronization is required before they begin parallelizing the simulation.

6.5 Course-Wide Pre- and Post-Course Surveys

Course-wide surveys shows that on seven-point scales, the mean self-reported experience across three PDC concepts increased from 1.62 to 2.21, while the mean across ten C++ programming topics increased from 4.17 to 5.26. The largest differences between aggregate programming-item means were for structures ($\Delta M = 2.73$) and classes and objects ($\Delta M = 1.93$). For each PDC experience item, the proportion selecting a rating of 5 or higher was 0% before the course and 25% afterward.

Table 6.3: Summary of PDC activities integrated into the CS1 course.

Activity	Approx. Time	Week	Relevant PDC Topics	Relevant CS1 Topics
Coin Search (Unplugged)	20 min	8	Sequential vs. parallel execution, workload decomposition, task independence	Linear search, loops
Flag Coloring Animation	30 min	8	Task decomposition, concurrency, synchronization, dependency	Problem solving
Parallel Search Animation	10 min	8	Data partitioning, embarrassingly parallel problems	Arrays, searching, loops
Parallel Sum Demonstration (OpenMP)	25 min	14	Data parallelism, reduction, compiler directives, speedup	Arrays, loops, functions
Odd-Even Parallel Sort Demo	30 min	14	Synchronization, independent tasks, parallel algorithms	Sorting, arrays, functions
Zombie Attack Assignment	1–2 weeks	15–16	Domain decomposition, scalability, locality, performance reasoning	2D arrays, simulation, nested loops, functions
Performance Discussion	15 min	16	Execution time, benchmarking, speedup, efficiency	Algorithm analysis, timing

Attitude ratings showed a different pattern. Mean interest in PDC was lower after the course (6.00 versus 4.50), while self-reported understanding of PDC remained unchanged at 4.00. Post-course helpfulness ratings were highest for programming assignments ($M = 6.62$), attending class ($M = 6.50$), and in-class practice ($M = 6.12$). These ratings concern the course as a whole rather than individual PDC activities.

The results describe aggregate self-reported experience and perceptions. Because responses were analyzed as unmatched groups, the differences do not establish individual learning gains or effects attributable to the PDC activities. Detailed results and analysis procedures appear in the appendix.

6.6 Other Activities and Ideas

Table 6.3 summarizes the plugged and unplugged activities used to introduce PDC concepts in my CS1 course, along with their approximate instructional time. Not all activities are used in every semester; the selection depends on the course schedule and instructional priorities. In practice, animations are used more frequently than unplugged activities because they convey the same concepts while requiring less class time. The reported durations are approximate and may vary depending on the level of student interaction, discussion, and the extent to which the instructor expands the activities with hands-on exercises and demonstrations.

Instructor Support Community One important future direction is the development of a broader community of instructors adopting introductory PDC teaching modules. A shared community could support the exchange of assignments, animations, classroom activities, datasets, and teaching experiences, particularly for faculty at teaching-focused institutions who may not have formal training in parallel computing.

Using Animations Unplugged activities are often effective for introducing abstract ideas such as concurrency, synchronization, and workload decomposition. However, many of these activities can feel overly simplistic or time-consuming for CS1 or CS2. Visual animations may provide a useful middle ground by retaining the intuitive nature of unplugged activities while allowing instructors to introduce concepts more efficiently during class. Developing a shared repository of lightweight educational animations could therefore be valuable to the broader PDC education community. The author also plans to continue developing additional interactive animations and visual teaching materials.

The Need for a Textbook Modern computing topics can extend the scope of traditional introductory programming curricula. Topics such as parallelism, simulations, and high-performance computing exposed students to contemporary computing challenges and applications; the anonymous survey summaries do not establish an increase in engagement. At the same time, early undergraduate students still require strong foundational instruction and structured textbook support before they can meaningfully engage with more advanced PDC concepts.

Prerequisite Gaps Introducing PDC concepts in CS1/CS2 courses remains challenging because many students lack the mathematical maturity and programming background typically associated with traditional parallel computing courses. Some PDC topics also do not naturally align with the programming languages commonly used in introductory courses. For example, while C++ is effective for teaching memory management, arrays, and performance-oriented programming, it provides relatively limited support for event-driven programming compared to languages and frameworks designed specifically for asynchronous or reactive systems.

Bridge to the Webster CS2 Pathways The conceptual foundation developed in this CS1 sequence supports the two Webster CS2 pathways presented in Chapters 15 and 16. Both build on students' introductory programming experience to develop further PDC concepts. Chapter 16 uses Python multiprocessing, parallel image processing, and performance engineering to extend the decomposition and performance ideas introduced here.

6.7 Conclusion

The modules mentioned above required minimal mathematical background and were designed to fit naturally within the existing C++ CS1 sequence without major restructuring. Student feedback provides descriptive information about reported familiarity and perceptions of course activities; the anonymous, unmatched survey responses do not establish changes in interest, confidence, or learning. Future work includes expanding the collection of animations and reusable teaching modules and exploring additional ways to integrate modern performance-oriented computing topics into introductory programming education.

Appendix

GitHub Resources and Teaching Materials

CS1 Webster Repository

Relationship to the CS2 Chapters. Chapters 15 and 16 describe the two Webster CS2 pathways that extend the introductory concepts developed here. Python multiprocessing, parallel image processing, and the Python Zombie assignment belong to Chapter 16 and are not part of this chapter's C++ CS1 resource list. The linked-list animation is also omitted from this CS1 list because its description concerns CS2 data structures.

Detailed Syllabus Comparison

Please see table 6.4. Detailed pre- and post- syllabus can be found here: syllabus

Table 6.4: High-level syllabus with PDC enhancements in CS1.

Traditional CS1 Topic	**PDC-Related Enhancement**
Introduction to Programming	Introduced the concept of parallel computing through unplugged activities and real-world examples of concurrency. (concepts only)
Loops and Iteration	Discussed independent iterations, data parallelism, and dividing repetitive tasks among multiple workers. (concepts, programming demo)
Arrays and Strings	Demonstrated data decomposition and partitioning of large datasets into smaller independent sections. (concepts, programming demo)
Searching and Counting Problems	Compared sequential and parallel approaches using classroom activities and simple performance measurements. (concepts, programming demo)
Grid-Based Simulations	Used zombie spread activities to illustrate parallel updates on large datasets. (concepts, programming demo, programming assignments)
Performance Measurement	Introduced basic timing experiments, speedup, and efficiency using small C++ examples. (in class demo)
Final Programming Assignments (Zombie Attack)	Included parallel-thinking exercises emphasizing decomposition, scalability, and workload distribution.

Organization of Material

Complete Teaching Repository This repository contains the instructional materials used throughout the module, including animations, assignments, and supporting input data files. course material Key materials include:

- **Link to unplugged activities:** unplugged activities. The following resources support the C++ CS1 activities in this chapter. To run an animation, download the HTML file and open it in a web
  - **Parallel Search Animation:** Parallel Search HTML source. Demonstrates workload decomposition and concurrent searching.

  - **Flag Coloring Animation:** Flag Coloring HTML source. Illustrates task decomposition and coordination.
  - **Zombie Attack Animation:** Zombie Attack HTML source. Illustrates infection propagation on a small grid before the programming assignment.
- **Parallel Programming slides**: instructional slides used in class. slides
- **Link to plugged activities:** plugged activities. Programs used in class
  - **Zombie Assignment**: Zombine assignment including input dataset
  - **Parallel Summation:** parallel summation. C++ program that shows how to calculate sum of a dataset
  - **Parallel Sort:** Parallel odd even sort. The program shows parallel odd-even sort.
- **Course schedule**: weekly schedule download as excel spreadsheet

Survey and Other Anonymized Data and Analysis

The analysis uses consenting responses identifying COSC 1560 from Spring 2025: eight pre-survey responses collected in February 2025 and eight post-survey responses collected in May 2025. Records from other courses or semesters, including the January multi-course record, are excluded. All reported items have eight valid responses at each applicable survey, except the post-survey C++ programming-experience items, which have seven valid responses each. Missing item responses are excluded from the corresponding item calculations; no values are imputed. Means and percentages were checked against the original survey exports.

Survey responses were coded on seven-point ordinal scales. Experience responses range from 1 = no experience at all to 7 = extremely experienced; agreement responses range from 1 = strongly disagree to 7 = strongly agree. Helpfulness responses range from harmful to helpful, with 4 = undecided, 5 = slightly helpful, 6 = very helpful, and 7 = extremely helpful. Percent favorable is the percentage of valid item responses coded 5 or higher. The final percentage column in the activity and outcome tables reports responses coded 6 or 7. Means summarize these ordinal ratings descriptively.

The PDC composite is the equal-weight average of the three PDC experience item means. The programming-experience composite is the equal-weight average of ten topic means. The pre-survey has one functions item; the post-survey separates function calls and writing functions, whose two means are averaged to represent the functions topic. Differences are calculated from unrounded values and then rounded for display, so they can differ slightly from subtraction of the displayed means. Standardized effect sizes are not reported.

The pre- and post-survey responses are analyzed as unmatched groups. Optional names are not used to link responses, and no identifying information is reported here. Accordingly, pre/post differences describe aggregate self-reported ratings rather than individual learning gains.

Pre-survey respondents reported lower experience ratings for parallel and distributed computing (PDC) than for the core programming topics summarized below. The composite mean across the three PDC-related experience items was 1.62 on the pre-survey and 2.21 on the post-survey. The percentage of favorable responses was 0% on the pre-survey and

25% on the post-survey for parallel processing, distributed computing, and event handling. These descriptive differences do not establish changes within individual students or effects attributable to the PDC activities.

The largest descriptive differences in reported ratings concerned core COSC 1560 programming topics that support later PDC learning. The composite programming-experience mean across ten C++ topics was 4.17 on the pre-survey and 5.26 on the post-survey. The largest differences between aggregate item means were observed for C++ structures ($\Delta M = 2.73$), classes and objects ($\Delta M = 1.93$), arithmetic expressions ($\Delta M = 1.09$), pointers ($\Delta M = 0.98$), branching ($\Delta M = 0.84$), and arrays ($\Delta M = 0.80$). These programming topics also appear in the chapter's examples of data decomposition, memory organization, and task-based problem solving.

Post-survey computing-attitude ratings are summarized below. In the post-survey, 87.5% of respondents to each item rated their interest in computing as favorable, and 87.5% rated their general understanding of computing as favorable. For the PDC-specific attitude items, 50.0% gave favorable ratings for interest in PDC, 50.0% gave favorable ratings for the importance of PDC, and 37.5% gave favorable ratings for understanding of PDC. These ratings describe respondents' perceptions. Several post-survey attitude means were lower than the corresponding pre-survey means; because responses were unmatched, neither higher nor lower aggregate ratings establish individual change.

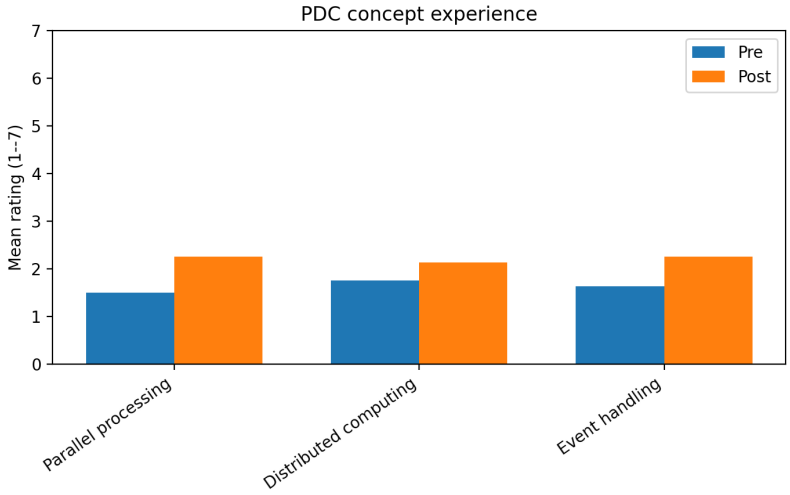


Figure 6.3: Pre/post comparison of student self-reported experience with PDC-related concepts. Mean ratings are reported on a seven-point scale. Responses analyzed as unmatched groups; pre $n = 8$, post $n = 8$.

Perceived helpfulness of course activities. Post-survey responses describe the perceived helpfulness of course activities (Table 6.8). Programming assignments received the highest rating ($M = 6.62$), with 100% of responses favorable and 100% rated very helpful or extremely helpful. Attending class received a mean rating of $M = 6.50$, followed by in-class practices ($M = 6.12$), textbook use ($M = 5.62$), exam preparation ($M = 5.62$), and meeting

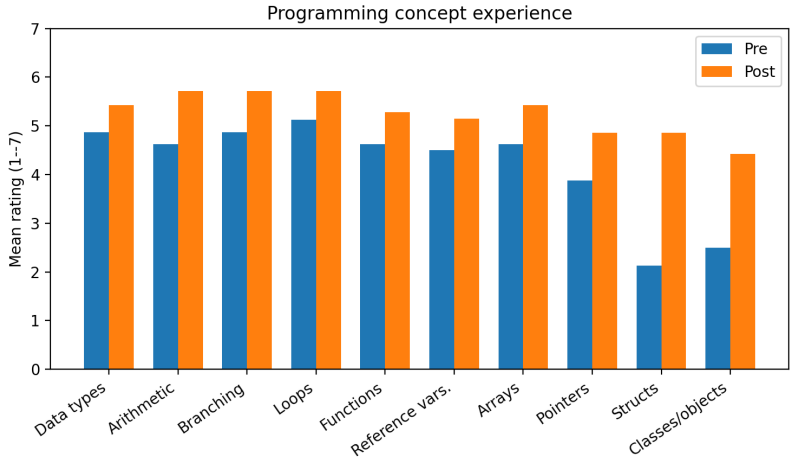


Figure 6.4: Pre/post comparison of student self-reported programming experience across core C++ topics. Responses analyzed as unmatched groups; pre item $n = 8$, post item $n = 7$.

Table 6.5: Descriptive pre/post differences in PDC-related experience. Percent favorable is the percentage of responses coded 5 or higher. Item-level $n = 8$.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp
Parallel processing	1.50	2.25	0.75	0.0	25.0	25.0
Distributed computing	1.75	2.12	0.38	0.0	25.0	25.0
Event handling	1.62	2.25	0.62	0.0	25.0	25.0

with the instructor outside of class ($M = 5.62$). These ratings describe perceived helpfulness of course-wide activities and do not isolate the effects of the PDC modules.

Self-reported course outcomes. Post-survey respondents rated their agreement with statements about COSC 1560 course outcomes (Table 6.9). All respondents to the corresponding items gave favorable ratings for writing C++ programs using arrays, pointers, structured data, and file operations; designing programs using department style guidelines; understanding the array–pointer relationship; reading and writing text/binary files; and using constructors and destructors. The highest post-course outcome means were for arrays/pointers/structures/files ($M = 6.12$), department style guidelines ($M = 6.00$), and array–pointer relationships ($M = 6.00$). These statements concern course-wide self-reported abilities rather than performance on a direct assessment of PDC knowledge.

Interpretation. These survey summaries describe students' reported experience and course perceptions. The unmatched self-reports do not establish individual learning gains or isolate the contribution of the PDC activities from other course instruction.

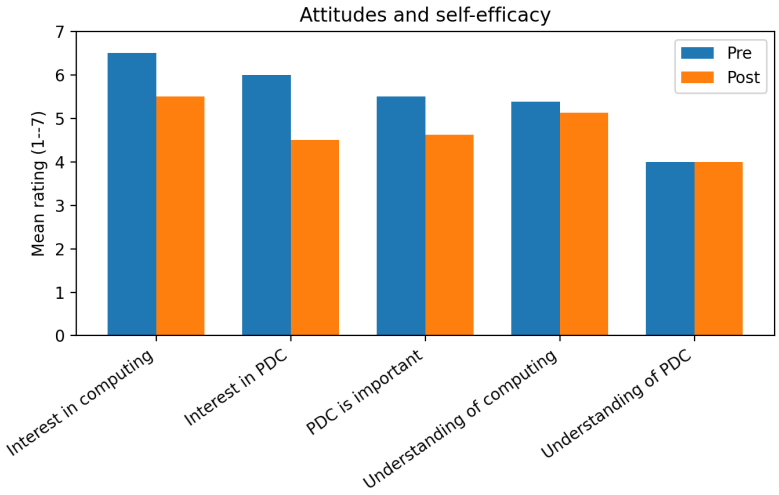


Figure 6.5: Pre/post comparison of student attitudes, interest, and self-reported understanding. Responses analyzed as unmatched groups; pre $n = 8$, post $n = 8$.

Table 6.6: Descriptive pre/post differences in programming experience across core C++ concepts. Pre item $n = 8$; post item $n = 7$. The post functions topic summarizes the function-calls and writing-functions items.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp
Data types	4.88	5.43	0.55	62.5	85.7	23.2
Arithmetic	4.62	5.71	1.09	62.5	85.7	23.2
Branching	4.88	5.71	0.84	75.0	85.7	10.7
Loops	5.12	5.71	0.59	87.5	100.0	12.5
Functions	4.62	5.29	0.66	62.5	85.7	23.2
Reference vars.	4.50	5.14	0.64	62.5	85.7	23.2
Arrays	4.62	5.43	0.80	75.0	71.4	-3.6
Pointers	3.88	4.86	0.98	37.5	57.1	19.6
Structs	2.12	4.86	2.73	12.5	57.1	44.6
Classes/objects	2.50	4.43	1.93	25.0	42.9	17.9

Table 6.7: Descriptive pre/post differences in interest, perceived importance, and self-reported understanding. Item-level $n = 8$.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp
Interest in computing	6.50	5.50	-1.00	100.0	87.5	-12.5
Interest in PDC	6.00	4.50	-1.50	100.0	50.0	-50.0
PDC is important	5.50	4.62	-0.88	75.0	50.0	-25.0
Understanding of computing	5.38	5.12	-0.25	87.5	87.5	0.0
Understanding of PDC	4.00	4.00	0.00	37.5	37.5	0.0

Table 6.8: Post-survey ratings of course learning activities. Item-level $n = 8$.

Activity	Mean	Favorable %	Very/Extremely Helpful %
Textbook	5.62	100.0	50.0
Programming assignments	6.62	100.0	100.0
In-class practices	6.12	100.0	87.5
Attending classes	6.50	100.0	100.0
Meeting with the instructor outside of class	5.62	62.5	62.5
Tutoring provided by REEG ARC peer tutors	4.38	25.0	12.5
Exams and preparing for exams	5.62	100.0	50.0

Table 6.9: Post-survey agreement with self-reported course-outcome statements. Item-level $n = 8$.

Learning outcome	Mean	Favorable %	Agree/Strongly Agree %
Arrays/pointers/structs/files	6.12	100.0	87.5
Department style guidelines	6.00	100.0	62.5
Array–pointer relationship	6.00	100.0	75.0
Character/string functions	5.38	87.5	50.0
Text/binary files	5.88	100.0	75.0
ADTs and structures	5.25	62.5	50.0
Classes with public/private members	5.62	87.5	62.5
Constructors/destructors	5.62	100.0	50.0

Chapter 7

CS1 Course Package – Casper College

Community College Adoption across Unplugged, OpenMP, Remote-Data, and Physical-Computing Activities[†]

Charlotte Gruner

Casper College, Casper WY

charlotte.gruner@caspercollege.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 200
7.1 Introduction . . . 202
7.2 How CS1 was changed and PDC topics integrated . . . 202
7.2.1 Integrated Syllabi (Pre and Post) . . . 202
7.2.2 Highlights . . . 203
7.2.3 Challenges . . . 204
7.3 New Unplugged Activities . . . 204
7.3.1 zyBooks Section 1.9 Parallel Computing . . . 204
7.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 204
7.3.1.2 Implementation Details . . . 204
7.3.1.3 Experience and Teaching Tips . . . 205
7.3.1.4 Data and Analysis . . . 206
7.3.2 Unplugged Penny Activity . . . 206
7.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 206
7.3.2.2 Implementation Details . . . 207
7.3.2.3 Experience and Teaching Tips . . . 207
7.3.2.4 Data and Analysis . . . 207
7.3.3 Unplugged Coin Addition Activity . . . 208
7.3.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 208
7.3.3.2 Implementation Details . . . 208
7.3.3.3 Experience and Teaching Tips . . . 209
7.3.3.4 Data and Analysis . . . 209
7.4 New Plugged-In Activities . . . 210
7.4.1 Plugged-In OpenMP Array Creation Activity . . . 210
7.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 210
7.4.1.2 Implementation Details . . . 210
7.4.1.3 Experience and Teaching Tips . . . 211
7.4.1.4 Data and Analysis . . . 211
7.4.2 Plugged-In OpenMP Array Sum Activity . . . 211
7.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 211
7.4.2.2 Implementation Details . . . 212
7.4.2.3 Experience and Teaching Tips . . . 213
7.4.2.4 Data and Analysis . . . 213
7.4.3 Plugged-In Distributed Data USGS API Earthquake Activity . . . 213
7.4.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 213
7.4.3.2 Implementation Details . . . 214
7.4.3.3 Experience and Teaching Tips . . . 214
7.4.3.4 Data and Analysis . . . 214
7.4.4 Plugged-In Physical Computing 1 Introduction and Traffic Light . . . 215
7.4.4.1 Problem Description, Prerequisites, and Learning Outcomes . . . 215
7.4.4.2 Implementation Details . . . 215
7.4.4.3 Experience and Teaching Tips . . . 216
7.4.4.4 Data and Analysis . . . 216
7.4.5 Plugged-In Physical Computing 2 Buttons and Multiple Processes . . . 217
7.4.5.1 Problem Description, Prerequisites, and Learning Outcomes . . . 217
7.4.5.2 Implementation Details . . . 217

7.4.5.3 Experience and Teaching Tips . . . 217
7.4.5.4 Data and Analysis . . . 218
7.4.6 Plugged-In Physical Computing 3 Temperature Sensor . . . 219
7.4.6.1 Problem Description, Prerequisites, and Learning Outcomes . . . 219
7.4.6.2 Implementation Details . . . 219
7.4.6.3 Experience and Teaching Tips . . . 220
7.4.6.4 Data and Analysis . . . 220
7.5 Course-wide Pre and Post Course Surveys . . . 221
7.6 Conclusion . . . 225
Appendix . . . 228

Chapter figures

7.1 Custom zyBooks Section 1.9 . . . 205
7.2 Custom zyBooks Section 1.9 continued . . . 206
7.3 CASPER CS1 Part 3 of Plugged-In OpenMP Array Sum Activity . . . 212
7.4 CASPER CS1 Hardware Circuit for Physical Computing Project 1 . . . 216
7.5 CASPER CS1 Physical Computing project 2 given code for instruction on the python multiprocessing package . . . 218
7.6 CASPER CS1 Sample Survey Question Reflecting on Level of Experience . . . 222
7.7 CASPER CS1 Sample Survey Question Reflecting on Level of Interest in Computing and PDC . . . 223

Chapter tables

7.1 CASPER CS1 Relevant PDC Topics and Bloom's Levels . . . 201
7.2 CASPER CS1 Baseline versus Summer 2025 PDC Intervention. . . . 203
7.3 CASPER CS1 Baseline versus SP26 PDC Intervention. . . . 203
7.4 CASPER CS1 Survey Results for Experience Questions . . . 221
7.5 CASPER CS1 Survey Results Experience with PDC Topics . . . 223
7.6 Survey Results for CS1 Interest Questions- Casper College . . . 223
7.7 CASPER CS1 Survey Results Interest Questions . . . 224
7.8 CASPER CS1 Survey Results for Understanding Questions . . . 224

Abstract

This chapter presents the integration of Parallel and Distributed Computing (PDC) concepts into an introductory computer science course (CS1) at Casper College, a two-year public institution. The primary goal was to embed foundational PDC topics into the existing curriculum without displacing traditional core content such as loops, arrays, functions, pointers, and classes. The pedagogical approach utilized a combination of unplugged activities, a custom digital textbook module, and plugged-in programming assignments in both C++ and Python. This chapter details two distinct implementations: one across an 8 week summer session and one spanning a full 16 week semester. Preliminary analysis of survey data collected in a baseline semester in which no PDC interventions were implemented and a semester in which several of the described interventions were implemented indicate that student perception of learning gains in non-PDC related topics remained generally the same while student perception of learning gains in PDC topics was higher, in general, in the semester in which PDC interventions were implemented.

Descriptor Elements

Programming Language Employed: predominantly C++, some Python (for Physical Computing projects, Spring 2026)

Textbook used: zyBooks[66] (Summer 2025) and REVEL edition of Starting out with C++ by Tony Gaddis[22] (Spring 2026)

Relevant core courses: CS1 (COSC 1030)

Pre-requisites: CS0 (COSC 1010) - this course is taught using Python/Scratch

Relevant PDC topics:

Relevant PDC topics and Blooms levels Table 7.1 lists the integrated PDC topics with Bloom's classification (Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR)).

Learning outcomes:

1. Define and recognize fundamental PDC terms such as multi-core, data parallelization, and task parallelization.
2. Apply OpenMP directives to parallelize array processing and analyze the resulting performance gains through benchmarking.
3. Identify and explain race conditions and shared resource conditions in both physical simulations and C++ code.
4. Implement and manage multiple processes in Python using the multiprocessing package for event-driven, task parallel, physical computing tasks.

Context for use: Casper College is a 2-year public, community college located in central Wyoming. The total enrollment at Casper College is typically between 4,500 students and 5,000 students. The most popular programs are in general studies, health care,

Table 7.1: CASPER CS1 Relevant PDC Topics and Bloom's Levels

PDC Concept	Chapter Section				
	7.3.1 zyBooks, 7.3.2 Unplugged Penny	7.3.3 Unplugged Coin Addition	7.4.1 Plugged-In Array Creation, 7.4.2 Plugged-In Array Sum	7.4.3 Plugged-In Earthquake	7.4.5 Phys Comp 2, 7.4.6 Phys Comp 3
PDC Vocabulary/Foundations	RE, UN			RE, UN	
Data Parallelism	RE		AP, AN		
Race Conditions/Synchronization		UN	UN, AN		
Speedup and Benchmarking			UN,AN	AP, AN	
Distributed Computing				UN, AP	
Event-Driven Programming					UN, AP
Task Parallelism	UN				AP, AN

and agriculture. The Computer Science Department, which is housed within the Mathematics Department in the School of Science, generally consists of around 50 declared majors with 80% working on freshman level coursework and 20% working on sophomore level courses. Demographically, Casper College has more students who are older than the typical 18–21 age group found at a typical university in the United States and less than half of the students are full time. Casper College also has a significant number of high school students enrolled in classes taught at their institutions (dual enrollment) and who attend classes taught at Casper College (concurrent enrollment). The CS1 course is taught at one of the local high schools by a high school instructor and on the Casper College campus by a college instructor. The CS1 course taught at the local high school was unaffected by the PDC interventions discussed in this chapter. The Computer Science Department has one full time faculty member. The CS1 course is a requirement for the A.S. degree in computer science and also a requirement for the A.S. degree in engineering. These are the two main populations of students who take CS1 at Casper College. The Computer Science Associates Degree at Casper College is meant to serve as the first two years of a 4-year degree program and includes the following core computer science classes: CS0 (COSC 1010) Introduction to Computer Science, CS1 (COSC 1030) Computer Science I, CS2 (COSC 2030) Computer Science II, (COSC 2150) Computer Organization, (COSC 2300) Discrete Structures. Class sizes for 1000-level courses tend to be below 20. Class sizes for 2000-level courses tend to be under 10. At Casper College the CS1 course, CS1 (COSC 1030), is classified as a 3 hour lecture, 2 hour lab class that transfers as a 4 credit class. This class is generally offered in two formats during the spring semester: an on-line, asynchronous format and in a live class format that meets every weekday for 50 minutes. During the summer semester, the class is offered in an 8-week online, asynchronous format.

7.1 Introduction

While integrating Parallel and Distributed Computing (PDC) early in the computer science curriculum is vital, doing so in the first two years without displacing traditional core topics remains a challenge because the curriculum in these classes is already packed and because there is a lack of curricular materials for this level that are easily incorporated into these classes. To address this, CDER initiated a two-year program to develop, test, and broadly disseminate adaptable curricular elements. Faculty teaching CS1 and CS2 at diverse institutions were recruited as "Testing Teams", supported by "Development Teams" who created and tested the classroom materials. Casper College participated in this effort as a Testing Team. This chapter explains Casper College's integration of modules adapted from the Development Team as well as new PDC materials into a traditional CS1 course. The overall goal of this effort was to identify opportunities to add PDC content without substantially changing the existing curriculum. The PDC interventions were implemented in an 8-week summer 2025 session (with 4 students at the beginning of the course, 2 finishing) and a spring 2026 semester course (11 students at the beginning, 5 who finished). The spring 2025 course (19 students at the beginning of the course, 13 finishing) served as the baseline and had no PDC interventions. This chapter discusses the changes that were made to two different sections of CS1 during the intervention semesters. Both unplugged and plugged-in interventions were employed. Some activities were adapted from those previously developed by the Development Teams, and some were created by us.

7.2 How CS1 was changed and PDC topics integrated

In the summer 2025 session the textbook zyBooks[66] *Programming in C++* was the primary textbook. The PDC interventions that were added to this course consisted of:

- A new instructor-written module for the zyBooks text (see Section 7.3.1).
- Unplugged Penny Activity (see Section 7.3.2).
- Plugged-In Earthquake Activity(see Section 7.4.3).

In the 2025 and 2026 spring semesters, the primary textbook was the REVEL edition of Starting out with C++ by Tony Gaddis[22]. For this course the main PDC interventions added in the spring semester 2026 were:

- Unplugged Penny Activity (see Section 7.3.2).
- Plugged-in OpenMP Array Creation Activity (see Section 7.4.1).
- Unplugged Coin Addition Activity (see Section 7.3.3).
- Plugged-in OpenMP Array Sum Activity (see Section 7.4.2).
- Physical Computing (see Sections 7.4.4-7.4.6).

7.2.1 Integrated Syllabi (Pre and Post)

A summary of the summer 2025 PDC material added and the timing of the material relative to the baseline CS1 material is shown in Table 7.2.

Table 7.2: CASPER CS1 Baseline versus Summer 2025 PDC Intervention.

Week	Baseline material	PDC interventions SU25
1	Introduction to C++, variables and assignment (zyBooks Ch 1,2)	Parallel Computing Intro (7.3.1), Unplugged Penny Activity (7.3.2)
2	Branches (zyBooks Ch 3)	—
3	Loops (zyBooks Ch 4)	—
4	Arrays/Vectors (zyBooks Ch 5)	—
5	Functions (zyBooks Ch 6)	Plugged-In Earthquake Activity(7.4.3)
6	Object/Classes (zyBooks Ch 7)	—
7	Pointers (zyBooks Ch 8)	—
8	Final exam week	—

A comparison of the baseline course from spring 2025 to the course with interventions in spring 2026 is shown in Table 7.3.

Table 7.3: CASPER CS1 Baseline versus SP26 PDC Intervention.

Week	Baseline (SP25)	PDC Interventions (SP26)
1	Intro to computers and programming/Environment set up (Gaddis Ch 1)	—
2	Intro to C++/on going project work (Gaddis Ch 2)	Physical Computing #1 (7.4.4)
3	Expressions and Interactivity (Gaddis Ch 3)	Physical Computing #2 (7.4.5)
4	Decision Structures (Gaddis Ch 4)	—
5	Loops and Files (Gaddis Ch 5)	—
6	Function (Gaddis Ch 6)	—
7	Arrays and Vectors (Gaddis Ch 7)	—
8	Arrays and Vectors (Gaddis Ch 7)	—
9	Pointers (Gaddis Ch 9)	Physical Computing #3 (7.4.6)
10	Pointers (Gaddis Ch 9)	Unplugged Penny Activity (7.3.2), Plugged-In OpenMP Array Creation Activity (7.4.1)
11	Structs (Gaddis Ch 11)	Plugged-In OpenMP Array Sum Activity (7.4.2)
12	Structs (Gaddis Ch 11)	—
13	Classes (Gaddis Ch 13)	Unplugged Coin Addition Activity (7.3.3)/Plugged-In OpenMP Array Sum Activity (7.4.2)
14	Classes (Gaddis Ch 13)	—
15	Searching/Sorting(Gaddis Ch 8)	—

7.2.2 Highlights

The same core CS1 material was covered in all the CS1 courses. In general, we were able to add PDC-related activities to weeks in the course that had a traditional CS1 topic that related to a PDC topic naturally. We also substituted project work for PDC-related project work. In the no-PDC-intervention of Spring 2024, the project work was a student-directed group project involving a Raspberry Pi GPIO controlled hardware In the PDC-intervention group this project was replaced with directed Physical Computing assignments that exposed students to topics in event handling and task parallelism. In the summer 2025 course, the project work was not done due to several logistical challenges.

7.2.3 Challenges

Several challenges were encountered in the process of adapting and creating PDC activities for use in this CS1 course. The first major challenge is that the material was unfamiliar to the instructor. Time to understand and work through the material was necessary and then additional time was needed to adapt the material to the particular context of the Casper College CS1 course. One critical adaptation challenge concerned adapting the unplugged activities for on-line students. These activities have, in general, been developed for classes where students are physically in class. Another significant challenge is that the OpenMP-based plugged-in materials have a steep learning curve that requires significant scaffolding, explanation, and sample code. The physical computing activities that were developed by us at Casper College also had a steep learning curve for students because of the unfamiliarity of not only the PDC related topics of event handling, task parallelism, and the Python `multiprocessing` package but also with the hardware elements used and the APIs needed to work with these elements.

7.3 New Unplugged Activities

7.3.1 zyBooks Section 1.9 Parallel Computing

7.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description Instructors can customize zyBooks by creating new sections and new assessments. In the summer semester of 2025, the primary text book was zyBooks, Programming in C++ [66]. In the first introductory chapter of this text, section 1.8 is titled "Computer Tour" which has content related to computer hardware. Since this section discusses the concept of a "processor", we modified TNTECH's introductory slides on PDC used for Unplugged Penny Activity (Section 2.3.1) to create a section 1.9 in zyBooks called "Parallel Computing" containing content about multiple processors and other concepts related to PDC.

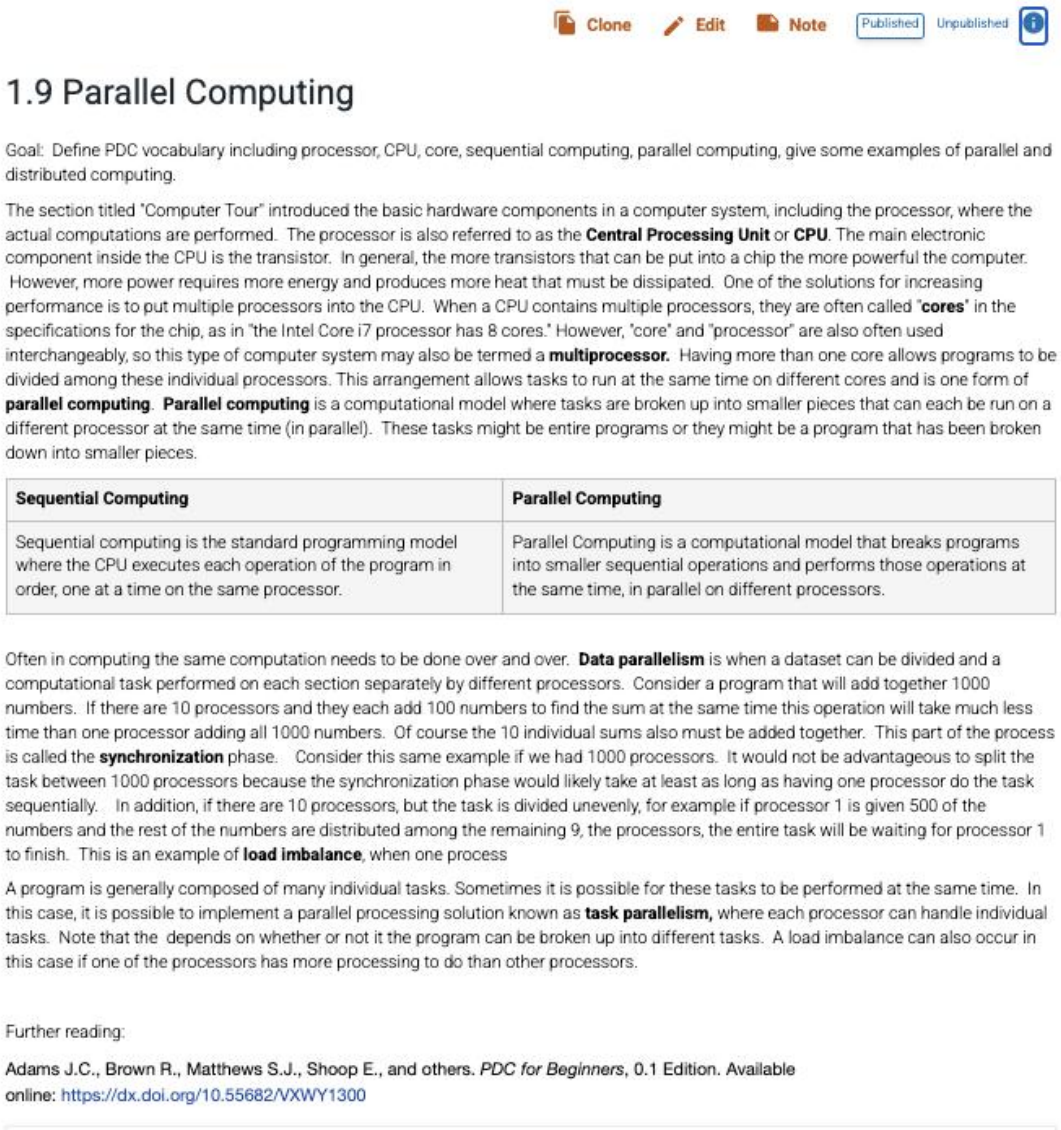
Prerequisites Knowledge of basic computer hardware, parts of a computer.

Learning Outcomes Students will be introduced to PDC vocabulary including "sequential computing", "parallel computing", "core", "multiprocessing", "data parallelism", "synchronization", "load imbalance", and "task parallelism."

7.3.1.2 Implementation Details

- **Logistics (Materials required for the activity, etc.)** This zyBooks module can be shared by the author with any interested instructors who are using zyBooks by emailing the instructor at charlotte.gruner@caspercollege.edu. The estimated time for students to complete the section is under 30 minutes.
- **How to Conduct** Students were assigned to read the section along with the rest of the chapter reading and complete the formative assessments in the form of participation activities within the chapter.

- **Instructional material (Lectures, Videos, Assignments, etc.)** A hardcopy of the materials can be found here.
  Figure 7.1 and Figure 7.2 show the custom zyBooks material to introduce topics in Parallel Computing.



Clone Edit Note Published Unpublished

1.9 Parallel Computing

Goal: Define PDC vocabulary including processor, CPU, core, sequential computing, parallel computing, give some examples of parallel and distributed computing.

The section titled "Computer Tour" introduced the basic hardware components in a computer system, including the processor, where the actual computations are performed. The processor is also referred to as the **Central Processing Unit** or **CPU**. The main electronic component inside the CPU is the transistor. In general, the more transistors that can be put into a chip the more powerful the computer. However, more power requires more energy and produces more heat that must be dissipated. One of the solutions for increasing performance is to put multiple processors into the CPU. When a CPU contains multiple processors, they are often called "**cores**" in the specifications for the chip, as in "the Intel Core i7 processor has 8 cores." However, "core" and "processor" are also often used interchangeably, so this type of computer system may also be termed a **multiprocessor.** Having more than one core allows programs to be divided among these individual processors. This arrangement allows tasks to run at the same time on different cores and is one form of **parallel computing**. **Parallel computing** is a computational model where tasks are broken up into smaller pieces that can each be run on a different processor at the same time (in parallel). These tasks might be entire programs or they might be a program that has been broken down into smaller pieces.

Sequential Computing	Parallel Computing
Sequential computing is the standard programming model where the CPU executes each operation of the program in order, one at a time on the same processor.	Parallel Computing is a computational model that breaks programs into smaller sequential operations and performs those operations at the same time, in parallel on different processors.

Often in computing the same computation needs to be done over and over. **Data parallelism** is when a dataset can be divided and a computational task performed on each section separately by different processors. Consider a program that will add together 1000 numbers. If there are 10 processors and they each add 100 numbers to find the sum at the same time this operation will take much less time than one processor adding all 1000 numbers. Of course the 10 individual sums also must be added together. This part of the process is called the **synchronization** phase. Consider this same example if we had 1000 processors. It would not be advantageous to split the task between 1000 processors because the synchronization phase would likely take at least as long as having one processor do the task sequentially. In addition, if there are 10 processors, but the task is divided unevenly, for example if processor 1 is given 500 of the numbers and the rest of the numbers are distributed among the remaining 9, the processors, the entire task will be waiting for processor 1 to finish. This is an example of **load imbalance**, when one process

A program is generally composed of many individual tasks. Sometimes it is possible for these tasks to be performed at the same time. In this case, it is possible to implement a parallel processing solution known as **task parallelism,** where each processor can handle individual tasks. Note that the depends on whether or not it the program can be broken up into different tasks. A load imbalance can also occur in this case if one of the processors has more processing to do than other processors.

Further reading:

Adams J.C., Brown R., Matthews S.J., Shoop E., and others. *PDC for Beginners*, 0.1 Edition. Available online: https://dx.doi.org/10.55682/VXWY1300

Figure 7.1: First part of new custom zyBooks section 1.9 Parallel Computing

7.3.1.3 Experience and Teaching Tips

In our experience, adding a section that included text, images and formative assessments in in the form of participation assignments to the zyBooks textbook by using previously created graphics and text was straightforward.

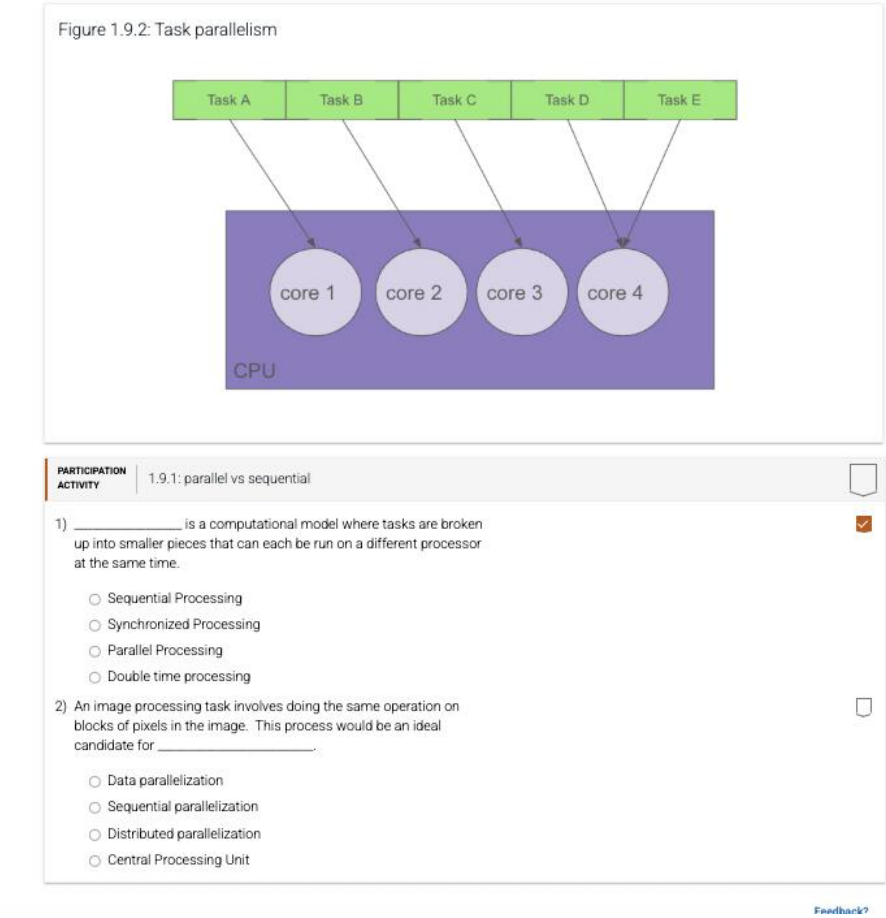


Figure 7.2: Second part of new custom zyBooks section 1.9 Parallel Computing

7.3.1.4 Data and Analysis

No formal quantitative data was collected on this specific activity. This intervention was implemented in a small on-line class with only two students. Students were not aware that this section was not part of the normal materials in the zyBooks text and read and completed the assessments seamlessly along with the rest of the chapter materials. This intervention would be easily scalable to a larger class size.

7.3.2 Unplugged Penny Activity

7.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description This activity is described in detail in section 1.3.2. We adapted TNTECH's materials described in section 2.3 for use in class and as an guided at home activity for on-line students. Briefly, a group of students are given the task of finding a specific penny from a group of pennies (such as the oldest). Students are guided toward increasing the speed at which this task can be done by employing more students in the group to perform the task to develop some intuition about parallelization and provide an

opportunity to introduce students to some vocabulary and concepts related to PDC.

Prerequisites No prerequisites are required for this activity.

Learning Outcomes

Students will be able to:

- explain the difference between sequential and parallel processing.
- describe how synchronization steps ensure a final, correct output when multiple workers are involved.
- recognize instances where the overhead of communication and synchronization takes more time than the actual work, negatively affecting performance.

7.3.2.2 Implementation Details

- **Logistics (Material required for the activity, etc.)** For the live class approximately 50 pennies per group (of 4-5 students) is required.
- **How to Conduct** Students are put into groups of 4-5 initially. In several steps, students are guided from constructing a sequential search where one student is timed in the task of finding the oldest penny to performing the task with multiple students participating, timing each experiment. Students should discover that adding more students to the task works for a while, but when taken to the extreme (each student having one penny) the task takes longer.
- **Instructional Material (Lectures, Videos, Assignments, etc.)** The adapted materials for on-line students are here.

7.3.2.3 Experience and Teaching Tips

We have found this unplugged activity is effective with groups ranging from high school students with no prior knowledge, CS0 students, CS1 students and CS2 students. With CS1 students we have used this activity successfully at two different times in the semester. In the first week of class, this activity fits with the introductory material on computer hardware (as it was used here in the summer semester 2025). When the hardware concept of processor is discussed, it is natural to extend the information that most modern computing systems have multiple processors. In Spring 2026 semester, we used this activity as the lead in to the first experience with OpenMP and parallelizing array initialization after arrays were introduced, which was approximately midway through the class.

7.3.2.4 Data and Analysis

No specific formal data was collected on this activity. Students were observed to have a high level of engagement in the activity which naturally brings up a variety of PDC topics. This activity is best for a group of students who are physically located in the same space. We have modified the activity directions for on-line students suggesting that they recruit family members or friends to participate, or alternately, to imagine how the activity would unfold with more people. On-line students could alternatively view a video of the activity.

It might also be possible for on-line synchronous students who are attending class remotely to participate if they are instructed to have some pennies available.

7.3.3 Unplugged Coin Addition Activity

7.3.3.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description. This unplugged activity simulates the addition of all elements in an array. The purpose of this activity is for students to recognize a situation in which there is a race condition because of a shared resource (the accumulator storage location in this example). This activity was designed to pair with the Plugged-In OpenMP Array Sum Activity (Section 7.4.1).

Prerequisites. It is helpful but not necessary if students have done the Unplugged Coin Search Activity. It is also helpful but not necessary that students have been exposed to array syntax to sum the contents of an array using an accumulation loop. It may additionally be motivating for students to have tried to naively parallelize this activity as shown below using OpenMP and discovered that the sum is not correctly calculated.

```
// THIS CAUSES A RACE CONDITION
#pragma omp parallel for
for (int i = 0; i < array_size; i++) {
    total += array[i]; // 'total' is shared and unprotected!
}
```

Learning Outcomes

Students will be able to:

- identify situations where parallel threads encounter a race condition due to a shared resource.
- describe why simple loop parallelization (like naive OpenMP directives) fails without proper synchronization and protection of shared resources.

7.3.3.2 Implementation Details

- **Logistics (Material required for the activity, etc.)** Enough containers (paper cups) with approximately 10 coins of mixed denominations (pennies, nickels, dimes, quarters) for each student participating. Ideally, there will be enough coins for each student to prevent a student from being able to look at the coins and immediately know how much they sum to, but not enough so that they will depart for the vending machine.
- **How to Conduct** Students should be in groups of 4-5. Each group should be given a cup with coins and 4 empty cups to use to distribute coins to team members for the parallelization. The problem is how to distribute the work of figuring out how much money is in the group. Students will recognize that dividing the coins among group members and having each student count their cup should decrease the time needed to

figure out the cumulative sum. However, students should be reminded that the loop to add the contents of an array, looks like:

```
total = 0
for each value in the array
        total = total + value
```

The important line to point out is that before the current element of the array can be added to *total*, the process needs to have access to the current value of *total*. To simulate this, have one student (or the instructor if the class is small) act as the keeper memory location for total, in charge of giving the total to each process (student) and recording the new value as each student accesses it. Each process needs to request the current contents of total, add the current value to it and then store it back in total. Students should observe that total is not being accumulated correctly as each student counter competes to get the current contents of total and then store their updated value.

- **Instructional material (Lectures, Videos, Assignments, etc.)** A video of the activity may be helpful for asynchronous online students, but synchronous on-line students may be able to participate if warned ahead of time to have coins available to use. Materials for this activity are found here.

7.3.3.3 Experience and Teaching Tips

So far, the instructor has tried this activity with in-class students in a CS1 class (Spring 2026) and a CS2 class (Fall 2025). Both times, the most difficult part was getting students to understand the directions surrounding obtaining and updating the *total* accumulator variable. Both class sizes were small, so the instructor acted as the storage location for this variable. Students emailed the instructor to request the value of the accumulator, the instructor sent back the value, the student did the addition and sent back the new value to store. The concept was to create an email trace that the class could follow after the experiment. Unfortunately, students were confused and only a couple of them understood what to do. Future attempts might include changing the instructions to include values written on a paper instead of coins, or a more complicated arithmetic that would slow down the students' tasks. Future attempts might also have the total stored on a small whiteboard that could be requested from the keeper of the accumulator. Even though this is work in progress, in both classes that this activity has been tried, students ultimately seemed to grasp the problem and were able to also realize that the simple solution of "locking" the accumulator so only one process could use it would not result in any performance gains, so they realize the solution would be for each process to have a separate subtotal and add those together after each process was done with it's part of the array.

7.3.3.4 Data and Analysis

No formal data was collected on this activity. In both classes students were engaged in the activity, especially the second time after the instructions were clarified. In both classes the students were ultimately able to figure out the issue with the shared resource and suggest a the solution of having processors computing partial sums.

7.4 New Plugged-In Activities

7.4.1 Plugged-In OpenMP Array Creation Activity

7.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description In this activity students are asked to write a C++ program that allocates space for a large array and then fills the array with the same value as the array index. The students are shown how to use the `#pragma omp parallel for` to parallelize the loop and are instructed to figure out how many different threads are created and which threads are handling which indices. The students are introduced to the use of the `chrono` library for timing code, so that the speedup can be documented. These materials were adapted from TNTECH's Plugged-In OpenMP Data Parallelism Activityin Section 2.4.2.

Prerequisites: Experience with C++ arrays and dynamic memory allocation using pointers.

Learning Outcomes

Students will:

- define and explain parallel computing terms: parallel processing, sequential processing.
- identify the number of cores on their particular machine.
- learn to use the OpenMP `#pragma omp parallel for` to do data parallelization
- learn how to compile using the `-fopenmp`.
- observe how the loop is split into chunks that are handled by different threads.
- use the C++ `chrono` library to measure and compare the execution time of sequential versus parallelized code.

7.4.1.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** Materials are found at here.
- **How-To Guides** In the materials created for this activity, students are guided in a step-by-step way toward learning about parallelizing a for loop using OpenMP and also guided toward use of the `chrono` library for benchmarking code. Students are encouraged to work in pairs to promote discussion as they work on this assignment.
  Part 1 of the assignment asks the students to do some preliminary research on PDC-related vocabulary and also some vocabulary specific to OpenMP to acquaint them with important terms. Students are allowed to converse with an AI about these topics and are encouraged to discuss these topics with their partner. Students also do an exploration to find out how many cores are on their own computer and on the classroom computers.
  Part 2 of the assignment students are instructed to write a sequential program to construct and initialize a large array. The goal of this task is for students to write a

program that takes a noticeable amount of time to complete. Prior to this activity, students may not have written any programs that did not appear to run almost instantly. This activity also requires students to use pointers to dynamically allocate memory, since there may not be enough memory for stack allocation of a large array. The next steps walk the students through thinking about whether or not this task could be done in parallel and how to implement this using the appropriate `#pragma` from OpenMP. In this step, the students are guided to use a smaller array so that they can document via print statements which threads are performing which array index initializations, which should ideally be the same as the number of cores that their computer has (which students should have determined in Part 1).
Part 3 of this assignment is for the students to learn to use the `chrono` library to benchmark the parallel and sequential code. They should find that there is speedup in the parallel version, but not as much as they predicted. Students are asked to reflect upon why the optimal speedup is not achieved. In this step, students are encouraged to spend some time using AI tools to learn about the `chrono` library to figure out how to use it for this application.

7.4.1.3 Experience and Teaching Tips

This exercise would work well as a 2 hour in-class lab. Another option would be for students to complete Part 1 of the activity before class so they would be prepared to start the coding part in class.

7.4.1.4 Data and Analysis

The Spring 2026 PDC Intervention CS1 course had only seven students. Three of the four in-person students completed the assignment successfully, reporting the time taken to do the assignment as 1.5 - 3 hours. We did give students in-class time to work on the assignment so that we could monitor and guide their progress. One of the three on-line student completed the assignment, reporting that it took 2.5 hours. The on-line student also requested that the instructions be clearer.

7.4.2 Plugged-In OpenMP Array Sum Activity

7.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description Students are instructed to write a C++ program to add all of the elements of the array and then to parallelize that loop using OpenMP, as learned in the Plugged-In OpenMP Array Creation Activity shown in Section 7.4.1. They will make the discovery that this does not work as desired and the accumulator (which is being shared by all the threads) is a shared resource that is experiencing a race condition. Then, students participate in the Unplugged Coin Addition Activity shown in Section 7.3.3 that is a demonstration of why the naive use of the `#pragma omp parallel for` may not work as desired.

Prerequisites Students should have already worked with arrays in C++ and be able to write a loop that sums the contents of an array. The Plugged-In OpenMP Array Creation

Activity (shown in Section 7.4.1) is the prerequisite; The Unplugged Coin Addition Activity (shown in Section 7.3.3) was designed as a co-requisite.

Learning Outcomes

Students will learn:

- to recognize a race condition.
- to recognize a shared variable.
- be able to describe why a simple loop parallelization using `#pragma omp parallel for` may fail

7.4.2.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** Repository with assignment and code is here.
- **How-To Guides** This activity is designed with multiple parts including: (1) student pair programming exploration, (2) an unplugged group activity (Unplugged Coin Addition Activity, Section 7.3.3), and (3) an instructor led demonstration.
  Part 1 of the assignment asks students to write and then parallelize code using `#pragma omp parallel for` to sum the contents of an array. Since at this point, they only know how to parallelize a for loop that does not take into account shared variables (as in the Plugged-In OpenMP Array Creation Activity from Section 7.4.1), they will try it this way and discover that this does not work, leading to questions about why this isn't working as desired.
  In Part 2, students participate in the Unplugged Coin Addition Activity (see Section 7.3.3) which will illustrate that the naive approach to summing the array with a shared variable won't work in parallel. Ideally students will realize that this code can be parallelized if each of the threads are responsible for finding a partial sum of part of the array and the accumulator that stores the final result must be protected and only accessed by one thread at a time.
  Part 3 of this activity is a demonstration explaining how to implement this idea using OpenMP. The key part of the code is shown in Figure 7.3

Figure 7.3: CASPER CS1 Part 3 of Plugged-In OpenMP Array Sum Activity

```
#pragma omp parallel //denotes the start of a parallel region
{

    long long int partialSum = 0;  //each thread will have it's own copy


    # pragma omp for  //use this inside a parallel region instead of "pragma omp parallel for"
    for (long long int j = 0; j < N; j++) {
            partialSum  += array[j]; //each thread computes a local sum
    }

    #pragma omp critical   //marks code that only one thread at a time can use
    {
            sum += partialSum;
    }
}
```

7.4.2.3 Experience and Teaching Tips

This activity could be split up over several class sessions (completing one of the parts each class, for example), or it could be done in one class period, especially if the class is at least 90 minutes. Part 1 of the activity could be done outside of class so that class time could be used for discussion. The unplugged group activity will be difficult for on-line students to replicate, so on-line students may need to watch a video of the live activity or if possible, they could likely participate synchronously but remotely via video conference.

7.4.2.4 Data and Analysis

We plan to refine this activity, as it was unclear whether students were fully achieving the intended learning outcomes. During implementation, students successfully wrote code to parallelize the array summation—which produced incorrect results—yet very few students demonstrated curiosity as to why the output was flawed. By contrast, this activity proved significantly more effective when piloted in a Spring 2026 CS2 course. Moving forward, the instructions could be restructured to guide CS1 students more explicitly toward identifying code failures and brainstorming potential underlying causes.

7.4.3 Plugged-In Distributed Data USGS API Earthquake Activity

7.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description This assignment was adapted from TNTECH's Plugged-In Earthquake Activity(see Section 2.4.1) in which students learn about distributed computing by developing a program that interacts with the USGS API to retrieve and process data related to seismic activity.

Prerequisites This assignment also assumes knowledge of cURL, JSON, and C++ functions. Students in our class were given mini-assignments on cURL and JSON before this assignment. (Theses preliminary assignments on cURL and JSON are in the GitHub repository for this assignment, located here.)

Learning Outcomes

Students will:

- explore how modern applications often rely on databases located on remote networks, requiring programs to interact with remote services to obtain information.
- learn what an API is and how to use it to request specific data.
- learn to use cURL as both a command line tool and a C++ library (libcurl) for transferring data from internet servers.
- **JSON structure** learn about JSON formatting in key/value pairs and use the json.hpp library to parse and manipulate the data within a C++ environment.

7.4.3.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** The materials used in this activity are available here
- **How-To Guides** In this assignment the students are given some sample code that shows how to use the NumbersAPI (`https://publicapi.dev/numbers-api`) to retrieve trivia about a number using the `curl.h` library along with a video explanation of the code. Next, students are given some sample code that uses the `json.hpp` library to create a JSON object that can be indexed with by the fields returned from the JSON request in the cURL. The end goal of this scaffolding is for students to extrapolate from the sample code to be able to learn about the USGS API and then to create a program that will allow a user to retrieve earthquake data for earthquakes above a user entered magnitude on their own. Since this was developed for an on-line class, it was designed to be a step-by-step activity that students could work through on their own, but these materials could also be used in a live class as a flipped classroom or paired programming lab assignment.

7.4.3.3 Experience and Teaching Tips

Because this chapter serves as an experience report of the pilot implementation, we document the real-world fragility of third-party public APIs. A major potential problem with any distributed data assignment is if the API changes or becomes unavailable the assignment won't work anymore. APIs need to be verified by the instructor before the assignment is given to students. For example, during the authoring period, the NumbersAPI used the example code was unavailable, so these materials may or may not work depending on whether or not the NumbersAPI is available. Rather than retroactively altering the historical assignment used by students, this outage highlights a critical lesson for CS1 adoption: external API dependencies introduce operational risks that instructors must evaluate prior to class. The USGS API appears more stable, but is more complex. The motivation for using the NumbersAPI was to have a slower introduction to the API material for the demonstration code as a scaffold so that students would be able to write the code that used the USGS API more independently. Another challenge in this activity is making sure that students are able to get the environment set up correctly. Details on this challenge are further discussed in Section 2.4.1.

7.4.3.4 Data and Analysis

Since these materials were used in an on-line class with only 2 students, there was limited and anecdotal data to report. Both students in the class completed the assignment satisfactorily and found it to be a challenging assignment. One student expressed enthusiasm about the real-world application.

In future iterations of this assignment, we will change the language of this assignment to Python and find a more stable API (other than NumbersAPI). Python has more accessible packages for distributed data that would not require discussion of cURL, for example, which for a CS1 audience, just increases the cognitive load without having a large benefit. JSON

would still need to be taught, but this topic flows smoothly from Python dictionaries. Exposure to Python would be a prerequisite to this re-worked assignment, but at Casper College most students have been exposed to Python in CS0, so this would not be a barrier. In fact, a deeper exposure to Python could be considered a benefit for students. This assignment might be a better match for a CS2 class, where students possess more programming sophistication. However, we were not able to figure out an appropriate time in the curriculum during the CS2 intervention semester of Fall 2025. A better fit for this content (at least in our context at Casper College) might be an introduction to Web App Development course. At Casper College, the Web App Development course has CS1 as a prerequisite, so students would have more sophistication, and the content seems to fit within the content Web App better than the content of CS1 or CS2. This solution would keep the overall goal of exposing students to concepts in PDC in the first two years of undergraduate computer science.

7.4.4 Plugged-In Physical Computing 1 Introduction and Traffic Light

7.4.4.1 Problem Description, Prerequisites, and Learning Outcomes

Description. This activity does not directly involve any PDC concepts, but it is required to complete the rest of the physical computing projects that do have PDC content. This project introduces students to using the GPIO pins to control a blinking light. Students are introduced to the `gpiozero` Python package (`https://gpiozero.readthedocs.io/en/stable/#`) that allows simplified access to the GPIO pins. Students are walked through the process of setting up a basic LED circuit on a breadboard and connecting it to the correct GPIO pin on the Raspberry Pi. The hardware set up is shown in Figure 7.4. Students are given some starter code that utilizes the `gpiozero` Python package to control the rate at which the LED in the circuit blinks. Students are then asked to take the project one step further by constructing a circuit and software that can control 3 LEDs to create a traffic light.

Prerequisites prior experience with Python.

Learning Outcomes

Students will:

- learn how to identify and use the 40 pins on a Raspberry Pi.
- learn about the pin types on a Raspberry Pi.
- practice setting up circuits using breadboard, jumper wires, resistors, and LEDs.
- learn how to use the `gpiozero` Python package to control hardware elements such as LEDs.

7.4.4.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** In addition to a Raspberry Pi, students will need a breadboard, 3 resistors, 3 LEDs of different colors,

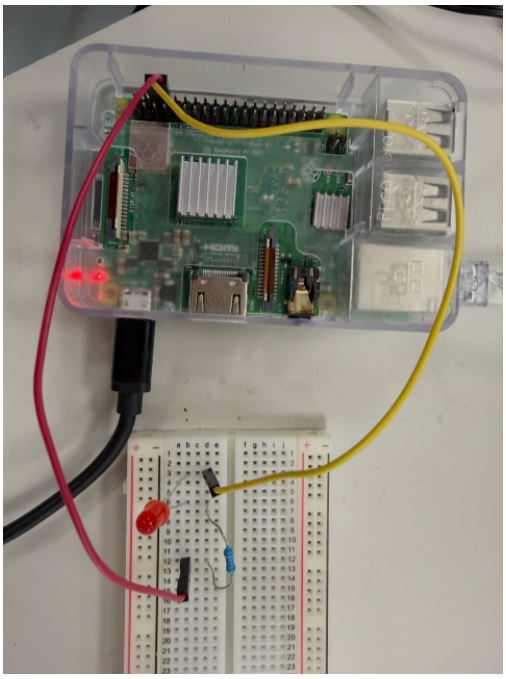


Figure 7.4: CASPER CS1 Hardware Circuit for Physical Computing Project 1

and jumper wires. We created kits that had all of these components for students to borrow. The complete directions are found here.

- **How-To Guides** Students who are unfamiliar with breadboards and electronic components will need explanation to be successful at this activity.

7.4.4.3 Experience and Teaching Tips

A concern with this activity is that the on-line students needed to overcome additional hurdles in acquiring the hardware kit and becoming familiar with the components, since they might not be at all familiar with circuitry.

7.4.4.4 Data and Analysis

This activity seemed to go pretty smoothly with most in-class students finishing in 20–30 minutes, with minimal instructor assistance. Students were engaged and seemed excited to be able to accomplish the task and see the results of their program running on the hardware. Only one of the three on-line students completed the assignment. We think the difficulty faced for the on-line students was likely due to the challenge of overcoming the inertia in starting a lab with many unfamiliar elements in both hardware and software. Student also needed to arrange to pick up the hardware kit itself. In the future, this issue will be mitigated by having all the hardware needed available to check out along with the Raspberry Pi all in one kit at the beginning of the semester. We are also planning additional video instruction to help all the students get started with physical computing.

7.4.5 Plugged-In Physical Computing 2 Buttons and Multiple Processes

7.4.5.1 Problem Description, Prerequisites, and Learning Outcomes

Description This project adds a button and code that monitors the button ("when pressed") and ("when released") so that when the button is pressed the LED is turned on and when the button is released the LED is turned off. The project guides students through changing the code so that when the button is pressed, the LED will continually flash a message in Morse code. All of the above tasks are possible without parallel processes. But if we want to change the program so that when we hit the button the message starts sending and when we hit the button again it stops, this requires multiple processes: one to send the message, one to monitor the button. The students are guided to use the Python multiprocessing package to manage these two processes. The code shown in Figure 7.5 is given in the assignment as an example of using the multiprocessing package in Python to manage multiple processes. The students are asked to take this work one step further by adding another button so that the resulting project will have button 1 result in message 1 from the LED and button 2 will result in a different message being transmitted by the LED.

Prerequisites Physical Computing 1 (Section 7.4.4).

Learning Outcomes

Students will:

- learn about event-driven programming by programming a physical button
- learn to manage multiple processes using the `multiprocessing` package to launch, monitor, terminate and join separate processes

7.4.5.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** In addition to a Raspberry Pi, students will need a breadboard, 1 resistor, 1 LED, 2 buttons and jumper wires. We created kits that had all of these components for students to borrow. The materials given to students and solutions are available here.
- **How-To Guides** This assignment is a multi-step walk through that guides the student toward building the starter code and circuit from scratch, and then asks the student to add on a new feature to an existing circuit. This assignment would work as a paired programming exercise, especially if more hardware-oriented students, (electrical and computer engineering students, for example) could be paired with computer science students to create multidisciplinary teams.

7.4.5.3 Experience and Teaching Tips

This assignment is definitely more difficult than Physical Computing 1 and required more time for students to complete. Since this activity builds on the skills introduced in Physical Computing 1, it is suggested that there should not be too much time between these

```python
import multiprocessing
from time import sleep
from gpiozero import LED, Button
from signal import pause

red = LED(14)
button = Button(17)
process = None


def message():
    # code to transmit the Morse Code message
    while True:
     red.on()
     sleep(1)
     red.off()
     sleep(1)

def button_callback():
    global process
    print("Button  pressed")

    if process and process.is_alive():
        print("stopping message")
        process.terminate()
        process.join()
        red.off()
    else:
        print("launching message")
        process = multiprocessing.Process(target=message)
        process.start()

button.when_pressed = button_callback

print("Ready to run")
pause()
```

Figure 7.5: CASPER CS1 Physical Computing project 2 given code for instruction on the python multiprocessing package

two activities, so students can easily recall and use the skill and knowledge from Physical Computing 1. This activity also starts off with a similar circuit, so if students have the same breadboard set up ready to go from Physical Computing 1, this will save time.

7.4.5.4 Data and Analysis

In the spring semester 2025, all four students in the in-person class completed this assignment within two 50-minute class periods and one out of three on-line student finished the assignment (the same students who completed Physical Computing 1). As with Physical Computing, overcoming the inertia to get started on this project might have been the difficulty for the on-line students.

7.4.6 Plugged-In Physical Computing 3 Temperature Sensor

7.4.6.1 Problem Description, Prerequisites, and Learning Outcomes

Description In this project, students are guided to use a *DHT22* temperature and humidity sensor in a task parallel, event driven project. The use of the *DHT22* sensor necessitates the use of a virtual environment and also the need to learn the API for controlling the sensor.

Prerequisites Physical Computing 1 and 2 (Sections 7.4.4, 7.4.5).

Learning Outcomes

Students will:

- learn to use a virtual environment (venv)
- learn how to use the adafruit-circuitpython-dht library to read data from the DHT22 temperature and humidity sensor
- manage multiple processes running simultaneously with the multiprocessing package
- learn how to use a shared object so that separate processes can have access to the same object
- be exposed to error handling using try-except blocks

7.4.6.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** In addition to a Raspberry Pi, students will need a breadboard, 1 resistor, 1 LED, 1 buttons, a `DHT22` temperature and humidity sensor, and jumper wires. We created kits that had all of these components for students to borrow. Materials for this activity are found here, including student materials and the solution code.
- **How-To Guides** This activity has four parts. Part 1 of the assignment walks students through setting up the circuit to use the sensor. Students set up a virtual environment to install the needed libraries to control the sensor. None of the previous classwork that students have done at Casper College would have required a virtual environment, so this definitely needed some explanation. We also provided the test code to test the sensors. These sensors have a high failure rate, so it was important to find out early if the sensors are even working properly. There is also some sophisticated exception handling that is necessary for this sensor, so all of that code had to be developed, provided to students, and explained to the students in the materials. In Part 2 of the activity, students are guided through adding an LED into the circuit that will turn on when the temperature exceeds a programmer-defined temperature threshold value and turn off when the temperature goes below that threshold. In Part 3, students use the `multiprocessing` package so that the LED will be able to blink a message instead of having a solid light. In Part 4, students add a button to the circuit and learn how multiple processes can share a resource, in this case a temperature threshold value to be able to develop a hardware/software solution to the following scenario:

You are designing a system that can monitor the temperature in either a greenhouse or a data center. The system should allow a temperature threshold to be toggled by pressing a button. If the temperature threshold is set for the greenhouse, pressing the button should set the threshold for the data center. Presumably the threshold for the greenhouse is higher than the temperature for the data center. If the temperature is higher than whatever the current threshold is set to, the LED should blink out a custom pattern.

The resulting solution needs to have:

- A process that is monitoring the *DHT22* sensor for the temperature value.
- A process that is monitoring the button that will toggle the threshold value that triggers the led blinking function.
- A process that is running the led blinking function if the temperature is above the threshold value.
- The solution also needs to have a shared object that is storing a shared value (the threshold) since when the multiprocessing package is used, each process gets its own copy of the entire script using the global variables.

Because this code turned out to be a lot more complex than the first two projects, students are given starter code to complete instead of asking student to develop from scratch for this final part of the assignment.

7.4.6.3 Experience and Teaching Tips

This was definitely the longest of the Physical Computing assignments created and probably should be broken up into multiple class periods and/or a longer time frame to completion. Based on observations of student work during this pilot implementation, we estimate that this assignment will likely take students 2-4 hours to complete all the parts. One major problem with this project is the use of the DHT22 sensor. It is not completely reliable, and can be a source of frustration. One of our 5 sensors had to be replaced because it wasn't working at all. The sensor also doesn't have very fast response time. Holding on to the sensor with your hand will result in the temperature being raised, but even after you release the sensor, the temperature does not go down very fast, so testing programs is quite laborious. The Python multiprocessing package also had a steep learning curve for both myself and the students.

7.4.6.4 Data and Analysis

All four students who ultimately passed the class completed this assignment successfully. Because of the steep learning curve for the parts of this assignment, it might be better to break the assignment up into shorter individual assignments that better scaffold the experience. We had intended for students to do a final project on physical computing in which they chose a new sensor, learned about the API and implemented a task parallel, event handling project. Alternately, the difficulty in testing the sensor code in hardware might create an opportunity for students to develop tests that simulate sensor input in software before testing in hardware. Due to time constraints as well as an unexpected opportunity

for students that came up during this pilot semester, we ended up going a different direction for the final project that is referenced in the course materials.

7.5 Course-wide Pre and Post Course Surveys

Course-wide pre and post surveys were given to students in the Spring 2025 (No PDC interventions) and Spring 2026 (PDC interventions as described) semesters. The common survey questions used in this project are described in Section 1.5). As mentioned previously, all three classes discussed in this chapter were small. The Spring 2025 semester had 13 students with 3 on-line and 10 in the live class. Of this population, 5 students completed both the pre-course survey and the post-course survey and also gave consent for data to be used for this study. The Spring 2026 semester had 7 students with 3 on-line and 4 in the live class. In this class, 3 students completed both pre- and post-course surveys and gave consent. The survey was not given to the Summer 2025 class since there were only 4 students who started the class and that number quickly dwindled to 2 students. The results of the surveys are summarized in Tables 7.4, 7.5, and 7.8.

No noticeable decrease in perception of learning in non-PDC topics. An important question that should be asked is whether inclusion of PDC topics has resulted in a decrease in learning gains for the core CS1 topics. Figure 7.4 summarizes the survey results for questions that asked students about their experience using various programming languages and programming skills and concepts in their programs. A sample question is shown in Figure 7.6.

Table 7.4: CASPER CS1 Survey Results for Experience Questions

	No intervention			Intervention		
Experience	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Computer experience	5.60	4.40	-1.20	5.33	5.00	-0.33
C	1.00	1.00	0.00	1.33	3.33	2.00
C++	1.20	3.40	2.20	1.50	4.00	2.50
C#	1.00	1.00	0.00	1.33	3.00	1.67
Java	1.00	1.00	0.00	2.00	1.33	-0.67
Javascript	1.00	1.20	0.20	1.67	1.33	-0.33
PHP	1.00	1.00	0.00	1.67	1.33	-0.33
Python	2.80	3.80	1.00	4.33	5.00	0.67
Scratch	1.40	1.80	0.40	4.00	3.00	-1.00
Visual Basic	1.20	1.40	0.20	1.67	1.00	-0.67
other	1.40	1.00	-0.40	2.00	2.00	0.00
data types	2.20	4.40	2.20	4.00	5.67	1.67
arithmetic expressions	2.20	4.00	1.80	4.00	5.67	1.67
branching	2.60	4.20	1.60	4.67	5.67	1.00
loops	3.00	4.20	1.20	4.67	5.33	0.67
functions	3.00	4.20	1.20	4.33	5.33	1.00
reference variables	2.80	3.40	0.60	4.67	4.67	0.00
arrays	1.60	4.00	2.40	3.00	4.67	1.67
pointers	1.40	3.00	1.60	1.67	4.33	2.67
C++ structures	1.40	3.20	1.80	1.00	5.00	4.00
classes & objects	1.40	3.40	2.00	2.00	4.33	2.33

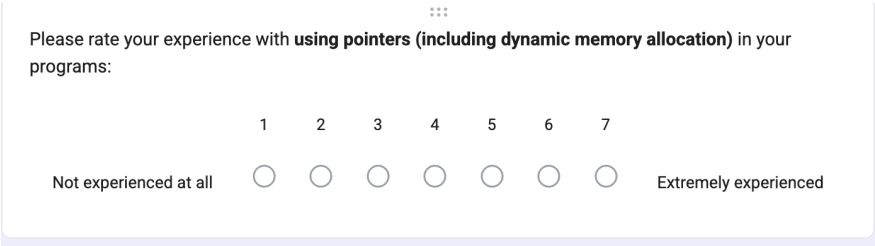


Figure 7.6: CASPER CS1 Sample Survey Question Reflecting on Level of Experience

The CS1 topics that were covered in both classes are shown highlighted in gray. The raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Spring 2025 semester are shown in the first two columns for questions in which students were asked to rate their experience with programming languages and the use of various programming topics in their programs. The mean of the difference (post-pre) of the matched survey responses are in the columns labeled $Mdif_base$ column. The number of responses for the baseline year with pre and post matches was 5. The last three columns show the responses and the mean difference for the Spring 2026 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 3. The rows highlighted in gray are CS1 topics from the survey that were presented in this course. Reassuringly, both the No PDC Intervention class and the PDC Intervention class rated themselves as more experienced in all of these topics except "reference variables", which showed the least gain in both classes and no gain in the PDC Intervention class, which may be due to the fact that this group also rated their pre experience higher than the No Intervention class. In fact, most of the topics covered in the class had a perceived gain of at least 1.0. In contrast, very few of the topics/languages that were not covered in the class showed gains, which is to be expected. The exception to this statement are:

- 1.67 gain reported in C# by the PDC Intervention class. Other researchers in this group have reported similar reported perceived gains in this category for C++ courses. Our hypothesis is that students are assuming that learning about C++ will translate into knowledge of C#. This group also reported higher levels of experience with C# in general, so it is also possible that this class gained some experience in C# through some other means during the course of the semester.
- 1.0 loss reported by the PDC Intervention class in the Scratch. This effect was due to one of the three students reducing their scores from 4 to 2, the other two students gave the same experience score for both pre and post. Scratch was not a language that students used in this class.

The summary of survey results that asked students about experience with PDC topics is shown in Table 7.5. In this table, raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Spring 2025 semester are shown in the first two columns for questions in which students were asked to rate their experience with programming languages and the use of various programming topics in their programs. The mean of the difference (post-pre) of the matched survey responses are in the columns labeled $Mdif_base$ column. The number of responses for the baseline year

with pre and post matches was 5. The last three columns show the responses and the mean difference for the Spring 2026 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 3. Encouragingly, the PDC Intervention class showed large increases in perceived experience level for each of the PDC topics covered. The No PDC Intervention class also indicated gains in these topics (even without the class interventions), but the magnitudes were smaller.

Table 7.5: CASPER CS1 Survey Results Experience with PDC Topics

	No intervention			Intervention		
Experience	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
parallel processing	1.00	2.00	1.00	2.00	5.00	3.00
distributed computing	1.00	1.20	0.20	1.33	5.33	4.00
event handling	1.00	1.80	0.80	3.00	5.00	2.00

Students were also asked to rate their level of interest in computing and in PDC. These survey questions are shown in Figure 7.7 and a summary of the results are shown in Table 7.6.

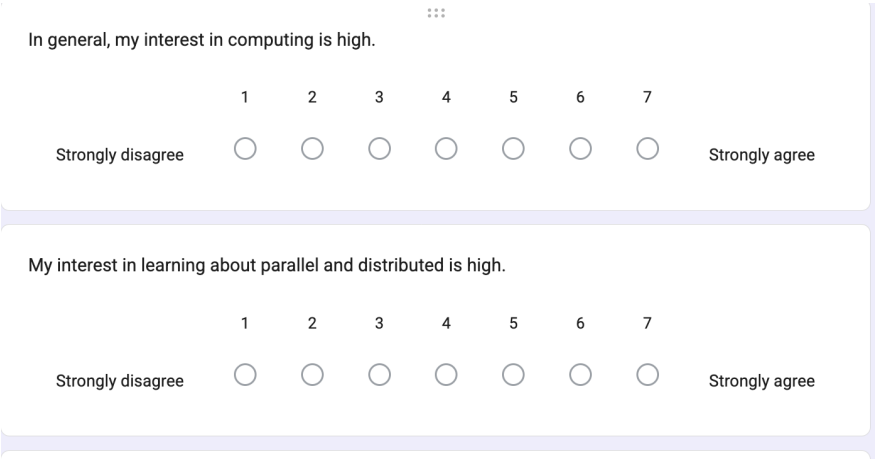


Figure 7.7: CASPER CS1 Sample Survey Question Reflecting on Level of Interest in Computing and PDC

Table 7.6: Survey Results for CS1 Interest Questions- Casper College

	No intervention			Intervention		
Interest	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
computing	5.40	5.20	-0.20	6.33	6.33	0.00
PDC	4.20	4.20	0.00	5.67	6.00	0.33

For Table 7.6, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Spring 2025 semester are shown in the first two columns for questions in which students were asked to rate their Interest in computing and interest in PDC topics. The mean of the difference (post- pre) of the matched survey responses are in the columns labeled $Mdif_base$ column. The number of responses

for the baseline year with pre and post matches was 5. The last three columns show the responses and the mean difference for the Spring 2026 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 3. These results show a generally high level of interest in both computing and PDC in both the pre and post surveys. A similar observation can be made about the results of asking students about their level of agreement with the statement "In general, I believe learning about parallel and distributed computing is important" as shown in Table 7.7.

Table 7.7: CASPER CS1 Survey Results Interest Questions

	No intervention			Intervention		
Importance	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
importance - PDC	4.00	4.75	0.75	6.00	6.00	0.00

For Table 7.7, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Spring 2025 semester are shown in the first two columns for questions in which students were asked the level of agreement with the following statement: “In general, I believe learning about parallel and distributed computing is important.” The mean of the difference (post-pre)of the matched survey responses are in the columns labeled $Mdif_base$ column. The number of responses for the baseline year with pre and post matches was 5. The last three columns show the responses and the mean difference for the Spring 2026 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 3.

Table 7.8: CASPER CS1 Survey Results for Understanding Questions

	No intervention			Intervention		
Understanding	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
computing	3.60	5.00	1.40	4.67	6.00	1.33
parallel and distributed computing	2.40	3.40	1.00	3.33	5.67	2.33

For Table 7.8, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Spring 2025 semester are shown in the first two columns for questions in which students were asked their level of agreement with the following statements: “I believe I have a good understanding of computing” and “I believe I have a good understanding of parallel and distributed computing.” The mean of the difference (post-pre)of the matched survey responses are in the columns labeled $Mdif_base$ column. The number of responses for the baseline year with pre and post matches was 5. The last three columns show the responses and the mean difference for the Spring 2026 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 3.

7.6 Conclusion

Lessons learned The experience of implementing PDC concepts in this chapter highlights a distinct trade-off between conceptual engagement and technical implementation hurdles. The unplugged activities, such as the Unplugged Penny Activity (Section 7.3.2) and Unplugged Coin Addition Activity (Section 7.3.3) were highly engaging to students. These exercises were also not difficult to prepare or implement and did not require a lot of class time. They were successful at building an intuitive foundation for fundamental PDC concepts and vocabulary. These activities generally rate lower in the Bloom's taxonomy such as "Remember" and "Understand". In our experience, a major difficulty in presenting PDC material at the CS1 level is transitioning to higher impact assignments that reach higher Bloom's levels such as "Analyze" or "Evaluate". Three sets of activities, the OpenMP-based activities, the Plugged-In Earthquake Activity, and the Physical Computing activities were used to attempt to move to higher Bloom's levels. The OpenMP based activities had a steep learning curve that necessitated the need for the instructor to provide large sections of code. This was also the case for the USGS Distributed Data assignment that was piloted in Summer 2025. This activity was dropped in the Spring 2026 semester CS1 course after it was used in the Summer 2025 course because it did not seem to fit within the Gaddis-based materials, it is a long assignment, and it requires many prerequisite activities. The physical computing assignments seem like a promising approach because they required students to completely break away from sequential programming. These assignments require managing multiple processes and event-driven tasks. Unfortunately, these assignments also proved to have a steep learning curve. The frustration of this experience was reflected in a blunt post-course survey response where one student labeled these assignments as "the one dud" of the course:

> While conceptually it would've been great for teaching parallelization, the unreliability of the raspberry pi devices, combined with using older text editors for a 'newer' language like python left me feeling like I was fighting the raspberry pi and learning more about electrical engineering than learning parallelization"

We believe that many of the deficiencies that were discovered in piloting the physical computing materials can be improved or eliminated, as will be discussed in the Future Work section.

Teaching Tips For instructors wishing to adopt these materials, we would advise starting with the "low hanging fruit" which are the unplugged activities. The unplugged activities work well, do not take a lot of time to prepare and are highly engaging to students. For the plugged-in activities, instructors need to make sure that student environments will work and that the instructors have taken the time to learn about the topic.

Future Work

The major challenge with these materials concerns the higher impact plugged-in activities. It is clear that more work will need to be done to better scaffold the plugged-in

assignments to reduce the technical overload while keeping the goal of reaching higher levels of Bloom's taxonomy.

OpenMP To improve the OpenMP-based work, one possibility is to change from using OpenMP to use a different library/API entirely. OpenMP has a steep learning curve and also an additional level of abstraction that can distract from the PDC concepts being taught. For example, in the OpenMP Array Creation Activity (Section 7.4.1), the use of the `#pragma omp parallel for` by itself does not give the programmer control over the number of threads. Although there are ways around this, fundamentally, this API is designed to make this judgment call for the program at runtime. OpenMP also abstracts away the concept of joining threads together, for example is not explicit in OpenMP. Maybe students should be starting with a library/API that is more predictable and more directly applies the fundamentals of PDC programming. Previous work done by Bunde [10] used `std::thread` class in C++ and `pthread` in C for instructional materials for CS2/Systems class. We plan to explore the possibility of using these libraries in place of OpenMP for the OpenMP Array Creation and OpenMP Array Sum Activity.

Plugged-In Earthquake Activity We plan to re-write the Plugged-In Earthquake Activityto use Python rather than C++. Our experience in this research project has shown that students, even at the CS1 level are not averse to changing programming languages for different applications. In our context at Casper College, most students have been exposed to Python in CS0 anyway, so this switch would not be difficult. The use of Python for this application would be an opportunity to demonstrate to students that matching the language to the application is desirable. As previously discussed, this activity may be better suited to CS2 or Web App Development.

Physical Computing As discussed previously, the physical computing activities have great potential to naturally introduce topics in event handling and task parallelism at higher Bloom's levels. In addition, these projects are a way to introduce students to applications that involve both hardware and software. We also feel that because of the added complexity of these activities as well as the more unusual nature of these activities, solutions will not be as readily available through interaction with AI tools, unlike many traditional CS1 activities. We have found that AI tools are unlikely to be able to provide complete, working solutions to students. This type of more complex assignment is an opportunity for students to use AI as a tool instead of a solution generator. To make these activities more accessible, it is clear that we are going to need to make significant revisions. Many revisions are straightforward and involve making sure the all the hardware needed is part of the materials checked out to students at the beginning of the semester. The other area of revisions that will be important is in better scaffolding activities and breaking up large activities into manageable pieces.

Addition of Materials to the Concurrent Enrollment High School Class The success of this pilot project has convinced us that many of these activities would be successful in the CS1 course that is offered as a concurrent class taught at our local high school. We

plan to encourage this course of action through yearly training that is part of our institution's discipline specific professional development for concurrent enrollment instructors.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter7-Casper

Detailed Pre and Post Syllabus

Baseline and Intervention Syllabi

Organization of Material

Materials in the Github repository are organized as follows;

- Detailed_Pre_and_Post_Syllabus – This directory contains information on the baseline semester of the course before the PDC interventions were added (Spring 2025) and the post semester when the PDC interventions were added (Spring 2026).
- Evaluation_Data_and_Analysis – This directory has documentation concerning the instruments and data collected during the course of this effort.
- Plugged_Activities – This directory contains resources used to implement the PDC Activities that involved coding activities.
  - Physical_Computing_1 – This directory contains the resources used to implement Physical Computing 1 Introduction and Traffic Light as described in Section 7.4.4.
  - Physical_Computing_2 – This directory contains the resources used to implement Physical Computing 2 Buttons and Multiple Processes as described in Section 7.4.5.

Survey and Other Anonymized Data and Analysis

Survey Data This directory contains aggregate data from the student perception surveys as well as the actual surveys.

Chapter 8

CS1 Course Package – Hawai'i Pacific University

Low-Preparation Unplugged and Plugged-in PDC Activities for Small Classes[†]

Mary L. Smith[1]

[1]Hawai'i Pacific University

mlsmith@hpu.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 232
8.1 Introduction . . . 235
8.2 How CS1 was changed and PDC topics integrated . . . 235
8.2.1 Integrated Syllabi (Pre and Post) . . . 235
8.2.2 Highlights . . . 236
8.2.3 Challenges . . . 237
8.2.4 Unplugged Activities . . . 237
8.3 New Unplugged Activities . . . 238
8.3.1 Unplugged - Flag Maker Activity . . . 238
8.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 238
8.3.1.2 Prerequisites . . . 239
8.3.1.3 Learning Outcomes . . . 239
8.3.1.4 Implementation Details . . . 239
8.3.1.5 Experience and Teaching Tips . . . 241
8.3.1.6 Data and Analysis . . . 241
8.3.2 Unplugged - Penny Activity . . . 244
8.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 244
8.3.2.2 Prerequisites . . . 245
8.3.2.3 Learning Outcomes . . . 245
8.3.2.4 Implementation Details . . . 245
8.3.2.5 Experience and Teaching Tips . . . 246
8.3.2.6 Data and Analysis . . . 247
8.4 New Plugged-in Activities . . . 250
8.4.1 Plugged-in - Parallel Sort Penny Activity . . . 250
8.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 250
8.4.1.2 Prerequisites . . . 250
8.4.1.3 Learning Outcomes . . . 250
8.4.1.4 Implementation Details . . . 251
8.4.1.5 Experience and Teaching Tips . . . 252
8.4.1.6 Data and Analysis . . . 253
8.5 Course-wide Pre and Post Course Surveys . . . 254
8.5.0.1 Course-wide Pre and Post Course Surveys Overall Conclusions . . . 257
8.6 Other Activities and Ideas . . . 259
8.6.1 Unplugged Activity - More Processors (MP) Writing Activity . . . 259
8.6.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 259
8.6.1.2 Prerequisites . . . 259
8.6.1.3 Learning Outcomes . . . 259
8.6.1.4 Implementation Details . . . 260
8.6.1.5 Experience and Teaching Tips . . . 261
8.6.2 Ideas on Activities . . . 261
8.7 Conclusion . . . 262
Appendix . . . 265

Chapter figures

8.1 Unplugged Flag Maker Activity Aspect Categories . . . 243
8.2 Penny Activities Aspect Categories . . . 248

Chapter tables

8.1 Bloom's Taxonomy Levels Addressed by Each PDC Activity 232
8.2 Syllabus Pre- and Post-PDC Topics Added 236
8.3 Estimated Instructional Time for Unplugged Flag Maker Activity 239
8.4 Pre-test and Post-test Performance by Question, Unplugged Flag Maker Activity $n = 6$ 242
8.5 Student Engagement Survey Results Grouped by adapted ASPECT Categories for the Unplugged Flag Maker Activity $n = 6$ 244
8.6 Estimated Instructional Time for Unplugged Penny Activity 245
8.7 Student Engagement Survey Results Grouped by Adapted ASPECT Categories for the Penny Activities, $n = 6$ 249
8.8 Estimated Instructional Time for Plugged-In Parallel Sort Penny Activity 251
8.9 Comparison of Pre- and Post-Course Survey Results (Fall 2024, $n = 5$) 255
8.10 Comparison of Pre- and Post-Course Survey Results (Fall 2025, $N = 4$) 256
8.11 Comparison of Pre- and Post-Course Survey Results (Spring 2026, $n = 5$) 257
8.12 Estimated Instructional Time for Unplugged MP Writing Activity 260

Abstract

This chapter describes how Hawai'i Pacific University (HPU) incorporated Parallel and Distributed Computing (PDC) concepts into sections of an introductory Computer Science I (CS1) course through unplugged and plugged-in learning activities centered on hands-on experiences that emphasized the PDC concepts being introduced. The activities were designed so that students did not need experience writing code in parallel. Instead, students first explored brief explanations of core PDC concepts and then participated in collaborative and interactive activities that allowed them to observe and discuss the concepts. These experiences reinforced student understanding of topics such as concurrency, message passing, shared memory, scheduling, load balancing, speed up and Amdahl's law while creating an engaging learning environment.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Starting Out With Java: Control Structures through Objects, 8th edition, Pearson Revel 2022, by Tony Gaddis.

Relevant core courses: The CS1 course includes CSCI 2911(3-credit lecture)/2916(1-credit lab)

Pre-requisites: None

Relevant PDC Topics and Bloom's Levels Per Activity: Table 8.1 summarizes the primary Bloom's Revised Taxonomy levels[1] addressed by each PDC activity. The unplugged activities emphasize the Remembering and Understanding levels, while the plugged-in Parallel Penny Sort activity additionally incorporates Applying through students' review and minor modification of existing Java programs.

Table 8.1: Bloom's Taxonomy Levels Addressed by Each PDC Activity

PDC Concept	**8.3.1 Flag Maker (Unplugged)**	**8.3.2 Penny (Unplugged)**	**8.4.1 Parallel Penny Sort (Plugged-in)**	**8.6.1 More Processors (Unplugged)**
Concurrency	RE, UN	RE, UN	UN, AP	RE, UN
Message Passing	RE, UN	RE, UN	UN, AP	RE, UN
Shared Memory	RE, UN	RE, UN	UN, AP	RE, UN
Scheduling / Load Balancing	RE, UN	RE, UN	UN, AP	RE, UN
Speedup / Amdahl's Law	RE, UN	RE, UN	UN, AP	RE, UN
Bloom's Revised Taxonomy Levels: RE-remembering, UN-understanding, AP-applying, AN-analyzing, EV-evaluating, CR-creating				

Activity Learning outcomes: The learning objectives for the activities presented in Sections 8.3.1, 8.3.2, and 8.6.1 primarily align with the Remembering (RE) and Understanding (UN) levels of Bloom's Revised Taxonomy, as shown in Table 8.1. These activities are designed to help students develop foundational knowledge of PDC concepts and build a conceptual understanding of how parallel systems operate. In contrast, the activity presented in Section 8.4.1 extends students' learning by moving beyond conceptual understanding and incorporating learning objectives at the Applying (AP) level of Bloom's Taxonomy. While students did not develop the code independently, they engaged in the Applying level of Bloom's Taxonomy by reviewing, interpreting, and making minor modifications to an existing Java programs, demonstrating how the concepts learned during the unplugged activity could be applied in a programming context.

Overall Learning outcomes: While each PDC activity has its own specific learning objectives, the activities collectively support the following overarching learning outcomes. Upon completion of the integrated PDC activities, students should be able to:

1. **Identify** fundamental PDC concepts, including sequential and parallel execution, data partitioning, task decomposition, load balancing, synchronization, communication overhead, speedup, scalability, and Amdahl's Law. *(Remembering)*
2. **Explain** how these foundational PDC concepts influence the design, execution, and performance of parallel programs, including the relationships among workload distribution, processor coordination, and overall execution time. *(Understanding)*
3. **Apply** foundational PDC concepts by interpreting and making minor modifications to Java programs that demonstrate parallel programming constructs, including Java's parallel libraries and the Fork/Join framework. *(Applying)*

Prior to each unplugged activity, the instructor discussed the importance of introducing PDC concepts early in the computer science curriculum. This was followed by an introductory overview of parallel computing and short videos that provided students with the background knowledge and context needed to engage with the concepts explored during the activity. After Activities 8.3.1, 8.3.2, and 8.6.1 the class held a post activity discussion to review the key insights gained. Participants compared how tasks were coordinated, analyzed their recorded times, and reflected on the challenges they encountered. The instructor guided the conversation, helping students connect their observations to learning objectives, defining parallel processing, explaining speedup, identifying factors that influence Amdahl's Law, understanding shared memory, and examining how parallel processing and effective task distribution improve efficiency. Through this discussion, participants also came to understand why adding more processors did not always lead to unlimited speedup.

Context for use: HPU is a private, mid-sized institution located in Honolulu, HI with a total enrollment of approximately 4,900 students, including approximately 3,500 undergraduates and 1,400 graduate students. The university is widely recognized for

its highly diverse student population, which includes learners from across the United States and more than 60 countries, contributing to a distinctly global campus environment. HPU provides a broad array of undergraduate and graduate programs spanning business, health sciences, natural sciences, liberal arts, and professional studies. The Department of Computer Science, located within the College of Natural and Computational Sciences, offers both a major and a minor in computer science, with small class sizes typically ranging from 5 to 20 students approximately 100 students currently pursuing the major.

8.1 Introduction

This chapter presents Hawai'i Pacific University's (HPU) integration of foundational parallel and distributed computing (PDC) concepts into an introductory Computer Science I (CS1) course comprising a three-credit lecture and a one-credit laboratory. The chapter describes the design, implementation, and classroom use of a series of unplugged, hands-on learning activities together with a complementary plugged-in programming activity that introduces students to fundamental PDC concepts early in the computing curriculum. The instructional activities are designed to foster conceptual understanding without requiring prior experience in parallel programming. Through active learning, students explore fundamental topics including concurrency, message passing, shared memory, scheduling, load balancing, speedup, and Amdahl's Law. Each activity combines a brief conceptual introduction with collaborative, hands-on exploration, providing students with an engaging and accessible foundation for understanding the principles of parallelism and their importance in modern computing systems.

The chapter is organized to present the integration of PDC concepts into the CS1 curriculum in a logical progression. It begins by describing the course modifications and syllabus updates made to incorporate PDC concepts while preserving the existing CS1 curriculum. The chapter then presents the unplugged and plugged-in instructional activities, including their learning outcomes, prerequisites, implementation details, teaching experiences, and evaluation results. It concludes with an analysis of the course-wide pre- and post-course surveys and a collection of additional activities and instructional ideas that educators can adapt to support future PDC instruction.

8.2 How CS1 was changed and PDC topics integrated

There were no significant changes made to the CS1 course; instead of removing existing CS1 topics, the instructor reduced the time previously spent on extended practice with concepts such as classes and class relationships, knowing these ideas would be revisited in greater depth in CS2. The course incorporated PDC concepts alongside the established curriculum rather than replacing or restructuring prior content. These concepts were introduced through a mix of lectures, class discussions, animated videos, and both unplugged and plugged-in lab activities. Typically, one lab session was devoted to these topics, allowing students to engage with the material through guided activities while still maintaining full coverage of the core CS1 content.

8.2.1 Integrated Syllabi (Pre and Post)

The overall structure and progression of the course syllabus remained largely the same before and after the introduction of PDC concepts. In both versions, the course begins with introductory programming content, including Java fundamentals, decision structures, loops and files, methods, text processing, regular expressions, classes, arrays and ArrayLists, testing with JUnit, and concludes with GUI development and a course wrap-up. The course continued to emphasize foundational programming knowledge and object-oriented design

principles. The primary change to the revised syllabus was the addition of a dedicated PDC Fundamentals unit near the end of the course, after students had developed sufficient experience with classes, arrays, and general problem solving in Java. In the earlier version of the syllabus, PDC concepts were not formally included and may have been referenced only briefly, if at all. Table 8.2 summarizes the differences between the pre-PDC and post-PDC CS1 syllabi, illustrating where PDC concepts and related instructional activities were integrated in the semester.

Table 8.2: Syllabus Pre- and Post-PDC Topics Added

Week	Pre-PDC Topics Covered	Post-PDC Topics Covered	PDC Activities
1	Chapter 1 – Introduction to Class; Introduction to Computers and Java	Chapter 1 – Introduction to Class; Introduction to Computers and Java	
2	Chapter 2 – Java Fundamentals; Decision Structures	Chapter 2 – Java Fundamentals; Decision Structures	
3	Chapters 2, 3 – Decision Structures	Chapters 2, 3 – Decision Structures	
4	Chapters 3, 4 – Decision Structures; Loops & Files	Chapters 3, 4 – Decision Structures; Loops & Files	
5	Chapter 4 – Loops & Files; Methods	Chapter 4 – Loops & Files; Methods	
6	Chapter 9 – Text Processing; Regular Expressions	Chapter 9 – Text Processing; Regular Expressions	
7	Chapter 5 – Methods	Chapter 5 – Methods	
8	Chapter 5 – Methods	Chapter 5 – Methods	
9	Break	Break	
10	Chapter 6 – Classes	Chapter 6 – Classes	
11	Chapter 6 – Classes	Chapter 6 – Classes	
12	Chapters 6, 7 – Classes; Arrays and the ArrayList Class	Chapters 6, 7 – Classes; Arrays and the ArrayList Class	
13	Chapters 7, 8 – Arrays and the ArrayList Class; Second Look at Classes	Chapters 7, 8 – Arrays and the ArrayList Class; Second Look at Classes; Test-Driven Development; JUnit Testing	
14	Chapters 8, 10 – Second Look at Classes, Inheritance	Chapters 8, 10 – Second Look at Classes, Inheritance, Introduction to GUI	
15	Introduction to GUI; Test-Driven Development; JUnit Testing	Introduction to PDC Fundamentals	One unplugged activity (Flag Maker, Penny, or the Multiple Processors (MP) Writing exercise) and one corresponding plugged-in activity (Parallel Sort Penny Activity).
16	Wrapping Things Up	Wrapping Things Up	Completion of the plugged-in activity (if not completed during Week 15).

8.2.2 Highlights

Most of the course content remained unchanged. The core programming topics, overall course pacing, and sequence of foundational Java concepts were preserved to maintain continuity with the existing curriculum and learning objectives. Students continued to progress through the same essential programming concepts in the same general order, ensuring that the course retained its emphasis on introductory programming competency and object-oriented development skills. Only minor modifications were required to integrate

PDC concepts. The primary change was the addition of a PDC Fundamentals unit to the syllabus. Rather than removing or substantially revising existing content, the integration was largely additive, since PDC concepts had not previously been covered in depth. Minor adjustments to scheduling and topic emphasis created space for the new material while preserving the existing curriculum. As a result, students were introduced to fundamental PDC concepts without disrupting the overall course structure or reducing coverage of essential CS1 programming topics.

8.2.3 Challenges

The implementation of PDC concepts introduced relatively few logistical or technical challenges because the instructional approach relied primarily on unplugged, discussion-based, and hands-on learning activities. The overall structure of the CS1 course, including programming assignments, laboratory exercises, and technology requirements, remained largely unchanged, making it straightforward to integrate the new content into the existing curriculum. No specialized software, hardware, parallel programming libraries, or distributed computing environments were required, allowing instructors to incorporate PDC concepts without placing additional technical demands on students or requiring substantial course redesign.

The primary challenge was pedagogical rather than technical. Because CS1 serves students with little or no prior programming experience, PDC concepts had to be introduced in a way that complemented the existing curriculum without overwhelming students with advanced implementation details. The unplugged activities were carefully designed to emphasize fundamental concepts such as parallel task execution, coordination, synchronization, communication, load balancing, speedup, and distributed problem solving through collaborative, hands-on experiences. A complementary plugged-in activity reinforced these concepts by illustrating how the same ideas could be expressed in Java, helping students connect their conceptual understanding to practical software implementations while avoiding the additional cognitive load of developing parallel programs.

From an instructional perspective, the greatest challenge was ensuring that PDC concepts enhanced rather than displaced core CS1 learning objectives. To address this, the activities were integrated alongside existing programming topics instead of replacing them, requiring only minor adjustments to the course schedule and a modest reduction in time previously devoted to extended practice on selected topics. Existing assignments, assessments, and laboratory activities required only minimal modification, allowing the instructor to maintain full coverage of the traditional CS1 curriculum while introducing foundational PDC concepts.

8.2.4 Unplugged Activities

An unplugged activity is an interactive, computer-free instructional exercise that uses physical materials and collaboration to illustrate fundamental computing concepts. Prior to incorporating coded examples, instruction focused primarily on unplugged activities to introduce core PDC concepts without the added complexity of programming syntax. These activities were typically preceded by an introductory discussion of PDC, followed by two short videos that provided foundational context. At least one unplugged activity continues

to be incorporated into the course. This hands-on instructional approach offered students an engaging way to explore fundamental concepts in parallel computing while reinforcing the advantages and challenges of multi-core processing. Each activity was structured to begin with a uniprocessor, sequential phase, followed by several parallel phases. Completion times for each phase were recorded, compared, and discussed upon conclusion of the activity. Because the activities shared similar instructional objectives and class time was limited, one or at most two activities were generally selected to cover the foundational PDC concepts. The primary unplugged activities used were the Unplugged Flag Maker Activity and the Unplugged Penny Activity. In the Unplugged Flag Maker Activity, groups of students colored a designated flag across multiple phases while simulating processor-based task distribution. In the Unplugged Penny Activity, students, again acting as processors, worked collaboratively to identify the oldest penny through three distinct phases, illustrating concepts such as task decomposition, workload distribution, coordination, and reduction.

Students demonstrated a high level of engagement throughout all of the unplugged activities, with the unplugged Penny Activity generating particularly strong enthusiasm. The activities fostered substantial student collaboration and were characterized by high levels of interaction, participation, and enthusiasm.

8.3 New Unplugged Activities

8.3.1 Unplugged - Flag Maker Activity

8.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

The Unplugged Flag Maker Activity is a hands-on instructional exercise that introduces introductory computer science students to foundational PDC concepts through collaborative, paper-based simulations in which students work together to color a flag using a variety of sequential and parallel execution strategies. Students progress from sequential execution to increasingly parallel execution scenarios, allowing them to develop an intuitive understanding of how multiple processors can work together to solve a problem. Throughout the activity, students explore core PDC concepts, including concurrency, data parallelism, speedup, scalability, load balancing, synchronization, mutual exclusion, and processor coordination. Designed specifically for CS1 students, the activity requires minimal technical prerequisites and no prior knowledge of parallel programming. Instead, students learn through active participation, collaboration, and guided discussion. During the activity, students assume the role of processors and use different parallel strategies to color the flag of Mauritius while coordinating their work to complete the task as efficiently as possible. By comparing multiple execution strategies, students observe how dividing work among processors can improve performance, how coordination influences execution time, and why synchronization is necessary when processors access shared regions. These experiences provide a conceptual foundation for later programming-based PDC activities.

8.3.1.2 Prerequisites

There are no prerequisites for this activity, making it suitable for students with little or no prior exposure to programming or PDC concepts.

8.3.1.3 Learning Outcomes

Upon completion of the activity, students should be able to:

- Explain the differences between sequential execution and parallel execution.
- Explain how partitioning data across multiple processors supports parallel processing.
- Describe how distributing work among processors contributes to effective load balancing.
- Identify the communication and synchronization overhead that can occur during parallel execution.
- Explain why the processor that finishes last determines the overall execution time of a parallel program.
- Explain how the activity demonstrates the concepts of speedup, scalability, and Amdahl's Law.

A complete description of the Flag Maker Activity, including the activity phases, facilitator instructions, student worksheets, discussion questions, and assessment materials, is provided in Section 1.3.1.

8.3.1.4 Implementation Details

The Flag Maker Activity was implemented during the Fall 2025 semester. Table 8.3 summarizes the estimated time required for each component of the activity, from the introduction and setup through execution, discussion, and optional assessments. These estimates serve as a general guideline and may vary depending on class size, student engagement, and the amount of discussion incorporated.

Table 8.3: Estimated Instructional Time for Unplugged Flag Maker Activity

Component	**Time (minutes)**
Pre-survey	3–5
Introduction and Instructions	5–10
Activity Setup	3–5
Activity Execution	20–25
Discussion and Debrief	6–10
Post-survey/Engagement Survey	3–5
Total Time	**40–60**

The activity is conducted in multiple phases, with each phase introducing a different execution strategy that illustrates a specific PDC concept. Before each phase begins, the instructor explains the objectives, reviews the rules, answers questions, and allows groups time to organize and assign roles appropriate for that phase. Students are typically organized into teams of five, although team sizes may increase to six depending on class enrollment.

Alternatively, smaller teams of two or three students may be formed initially and later combined for the parallel execution scenarios. Throughout the activity, students act as processors responsible for coloring individual "pixels" on the flag of Mauritius according to the assigned execution strategy. Each group receives four gridded flag templates one for each phase and a set of colored drawing tools. Once all groups are prepared, the facilitator signals the start of the activity, and all teams begin simultaneously.

The activity consists of four progressively more complex execution scenarios:

Phase 1 – Sequential (Uniprocessor) Execution: One student colors the entire Mauritius flag sequentially while another student records the execution time. This phase establishes a sequential performance baseline for comparison with the parallel implementations.

Phase 2 – Two-Processor Parallel Execution: Two students simultaneously color different portions of the flag by dividing the work by stripes, while a third student records the execution time. Because both processors share the same worksheet and drawing tools, this phase demonstrates task decomposition, parallel execution, communication, and coordination when accessing shared resources.

Phase 3 – Four-Processor Parallel Execution (Stripe Partitioning): Four students each color one stripe of the flag while another student records the execution time. Sharing the same worksheet and coloring tools introduces contention for shared resources and highlights the need for synchronization and coordination. The slowest processor determines the overall completion time.

Phase 4 – Four-Processor Parallel Execution (Column Partitioning): Four students simultaneously color different columns of the flag while another student records the execution time. Unlike the previous phase, each processor must use all four colors, creating additional opportunities for contention and coordination. This phase allows students to compare how different task partitioning strategies affect performance and resource utilization.

At the end of each phase, the instructor records and publicly displays each group's completion time to encourage discussion and comparison of the different execution strategies. Students then reflect on how increasing the number of processors, different task decomposition strategies, workload distribution, communication, synchronization, and contention for shared resources affected overall performance. These guided discussions reinforce the targeted PDC concepts before students progress to the next phase. Complete implementation instructions are provided in Section 1.3.1.

- **Materials Required** The Flag Maker Activity requires only inexpensive classroom materials, making it easy to implement in a variety of instructional settings. Required materials include:
  - Four sheets of gridded paper for each student group (one for each activity phase).
  - Colored drawing implements (e.g., markers, crayons, colored pencils, or bingo daubers) in red, blue, yellow, and green (or additional colors if a different flag is used).
  - A timer, stopwatch, or mobile phone timer.
- **Instructional material (Lectures, Videos, Assignments, etc.)**

8.3.1.5 Experience and Teaching Tips

The Unplugged Flag Maker Activity has been an effective way to introduce students to foundational PDC concepts, as it allows them to experience parallel execution through a familiar, visual task. Before each phase begins, allow groups a few minutes to discuss their roles and strategy. This planning period encourages students to think about how work should be divided among processors and naturally introduces concepts such as concurrency, scheduling, and data parallelism. Displaying each group's completion time after each phase also provides students with opportunities to compare performance and predict how increasing the number of processors affects overall execution time. Encourage students to reflect on the differences between the four phases rather than simply focusing on which group finishes first. The sequential phase establishes a baseline for comparison, while the two-processor and four-processor phases illustrate how dividing work among processors can improve performance. Comparing the two four-processor approaches (partitioning by stripes versus partitioning by columns) often leads to meaningful discussions about workload distribution, communication overhead, and why different decomposition strategies may produce different results even when the same number of processors is used. The shared materials used throughout the activity provide a natural demonstration of mutual exclusion and synchronization. Because processors share the same sheet of paper and colored drawing implements, students frequently encounter situations in which they must wait for access to a marker or coordinate their movements to avoid interfering with one another. Rather than preventing these interactions, instructors should use them as teachable moments to discuss contention for shared resources, synchronization, and the role of scheduling algorithms such as first-come, first-served (FCFS) in managing resource access. Students quickly recognize that adding processors alone does not eliminate bottlenecks when shared resources must be coordinated. Conclude the activity with a class discussion comparing the recorded execution times from each phase. Ask students which approach produced the greatest speedup, which strategy resulted in the best load balance, and what factors prevented perfect linear speedup. Encourage them to consider how communication, synchronization, waiting for shared resources, and differences in processor speed affected the overall completion time. These reflections help students connect their hands-on experiences to the underlying PDC concepts and prepare them to understand similar challenges in software-based parallel programming.

8.3.1.6 Data and Analysis

The evaluation of this activity incorporated both learning and engagement measures. Learning gains were assessed using a one-group pretest-posttest design in which students completed five multiple-choice questions designed to assess their understanding of key parallel computing concepts, including task decomposition, speedup, contention, scalability, and pipelining. Student engagement was assessed separately using a survey adapted from the Assessing Student Perspective of Engagement in Class Tool (ASPECT) [70]. The survey was administered immediately following the activity and the learning assessment. Table 8.4 summarizes student performance on the pre- and post- learning assessments. Students demonstrated improvements on the questions related to task decomposition and contention, while performance on speedup and scalability remained unchanged. Performance declined on the

pipelining question, suggesting that this concept may require additional instructional emphasis. Overall, the average score changed from 83.3% on the pre-test to 80.0% on the post-test. Given the small sample size (n = 6), these results are descriptive and are intended to identify tendencies rather than establish statistical significance.

Table 8.4: Pre-test and Post-test Performance by Question, Unplugged Flag Maker Activity $n = 6$

Question	Concept	Pre-test	Post-test	Change
Q1	Task Decomposition	83.3%	100.0%	+16.7%
Q2	Speedup	100.0%	100.0%	0.0%
Q3	Contention	33.3%	50.0%	+16.7%
Q4	Scalability	100.0%	100.0%	0.0%
Q5	Pipelining	100.0%	50.0%	−50.0%
Overall	**Average**	**83.3%**	**80.0%**	−3.3%

The student engagement survey was adapted from the ASPECT [70] and modified to align with the learning objectives of the PDC activity. In addition to the engagement questions, the survey included several demographic items to provide background information about participants. However, the analysis of demographic responses is not presented in this chapter due to the small sample size and the potential risk of identifying individual students. This decision protected participant confidentiality and avoided drawing conclusions about demographic subgroups with insufficient representation. The engagement survey questions were grouped according to the primary ASPECT categories for analysis, with two additional PDC-specific questions included to evaluate learning outcomes that were related to this instructional intervention. Figure 8.1 illustrates the survey questions grouped by category for the Flag Maker activity.

Table 8.5 summarizes the student engagement survey results grouped according to the ASPECT categories, along with two additional Curriculum Integration items included to evaluate how well the activity connected to the CS1 curriculum. Overall, students reported highly positive perceptions of the activity, with category mean scores ranging from 3.67 to 4.88 on a five-point Likert scale. These findings suggest that students found the activity engaging, well supported by the instructor, and beneficial for learning introductory PDC concepts and related programming topics. The highest ratings were observed for **Instructor Contribution** (M = 4.88), indicating that students perceived the instructor as well prepared, enthusiastic, supportive, and committed to their learning throughout the activity. These findings suggest that instructor facilitation played an important role in creating a positive and engaging learning environment. The **Personal Effort** category also received high ratings (M = 4.50), indicating that students remained focused during the activity, worked hard, and believed they made meaningful contributions to their groups. These responses suggest that the collaborative nature of the activity encouraged active participation and sustained student engagement. Students also responded favorably to the **Value of the Group Activity** (M = 4.41), indicating that they enjoyed the activity, found the group discussions beneficial, and believed the activity increased both their understanding of and interest in parallel computing. The high ratings suggest that students viewed the collaborative learning experience as a valuable component of the course and appreciated

Category and Survey Questions
Value of Group Activity*
• Explaining the material to my group improved my understanding of it.
• Having the material explained to me by my group members improved my understanding of it.
• Group discussions during the activity contributed to my understanding of parallel computing.
• I had fun during the activity.
• Overall, the other members of my group made valuable contributions during the activity.
• I would prefer to take a class that includes this group activity over one that does not.
• I am confident in my understanding of the material presented during the activity.
• The activity increased my understanding of parallel computing.
• The activity stimulated my interest in parallel computing.
Personal Effort*
• I made a valuable contribution to my group during this activity.
• I was focused during the activity.
• I worked hard during the activity.
Instructor Contribution*
• The instructor seemed prepared for the activity.
• The instructor put a good deal of effort into my learning from the activity.
• The instructor's enthusiasm made me more interested in the activity.
• The instructor was available to answer questions during the activity.
Curriculum Integration
• The activity increased my understanding of loops.
• I like that the activity tied into the class's current programming topics.

Note. The student engagement survey was adapted from the *Assessing Student Perspective of Engagement in Class Tool* (ASPECT) [1]. Several survey items were modified to reference the PDC activity and its learning objectives, and two PDC-specific questions were added to evaluate outcomes unique to this instructional intervention. The survey retained the three primary ASPECT categories noted by an asterisk—*Value of Group Activity, Personal Effort, and Instructor Contribution*—for analysis.

Figure 8.1: Unplugged Flag Maker Activity Aspect Categories

learning through peer interaction. The **Curriculum Integration** category received a positive overall rating (M = 3.67). Students generally agreed that the activity connected to the programming topics being covered in the course and supported their understanding of the current curriculum. Although this category received the lowest mean among the four categories, the ratings remained above the midpoint of the five-point scale. This finding is not unexpected, as the primary objective of the activity was to introduce PDC concepts rather than explicitly reinforce loops or other specific programming constructs. Nevertheless, the results suggest that students viewed the activity as relevant to the CS1 curriculum while indicating opportunities to strengthen the connections between the activity and the programming concepts taught in the course. Overall, the survey results indicate that the activity successfully engaged students, promoted active participation, and fostered positive perceptions of learning introductory PDC concepts (overall M = 4.44). Although the findings should be interpreted cautiously because of the small sample size ($n = 6$), they provide

encouraging descriptive evidence that collaborative, activity-based instruction can effectively support student engagement in an introductory computer science course. The open-ended survey responses further complemented these findings. When asked what they found most interesting, students highlighted fundamental PDC concepts, including race conditions, how multiple processors coordinate to solve problems, and the challenges associated with communication and synchronization among processors. These responses suggest that the activity successfully introduced students to core PDC concepts beyond traditional sequential programming. When asked how the activity could be improved, students offered only minor suggestions, such as using colored highlighters to better align with the activity format and providing a slide deck or PDF version of the presentation for students who did not have access to Microsoft PowerPoint. One student indicated that no improvements were necessary. Overall, the qualitative responses reinforce the quantitative survey results, demonstrating that students found the activity engaging, informative, and well-designed while identifying only minor opportunities for refinement.

Table 8.5: Student Engagement Survey Results Grouped by adapted ASPECT Categories for the Unplugged Flag Maker Activity $n = 6$

Category	**Items**	**Mean**
Value of the Group Activity (ASPECT)	9	4.41
Personal Effort (ASPECT)	3	4.50
Instructor Contribution (ASPECT)	4	4.88
Curriculum Integration	2	3.67
Overall	**18**	**4.44**

8.3.2 Unplugged - Penny Activity

8.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

The Unplugged Penny Activity is a hands-on instructional exercise that introduces introductory computer science students to foundational PDC concepts through a collaborative sorting-and-searching task. Students assume the role of processors and work individually or in parallel to identify the oldest penny from one or more coin collections. As the activity progresses, students compare sequential execution with multiple parallel execution strategies to observe how workload distribution and processor coordination influence performance. The activity requires no technical prerequisites and is intended for CS1 students with no prior experience in parallel programming. Through active participation, students develop an intuitive understanding of task decomposition, data partitioning, speedup, scalability, communication overhead, and load balancing. The activity also introduces the practical effects of balanced and unbalanced workloads, providing a concrete demonstration of Amdahl's Law and the factors that influence parallel performance.

8.3.2.2 Prerequisites

There are no prerequisites for this activity, making it suitable for students with little or no prior exposure to programming or PDC concepts.

8.3.2.3 Learning Outcomes

Upon completion of the activity, students should be able to:

- Explain the differences between sequential execution and parallel execution.
- Explain how partitioning data across multiple processors supports parallel processing.
- Describe how distributing work among processors contributes to effective load balancing.
- Identify the communication and synchronization overhead that can occur during parallel execution.
- Explain why the processor that finishes last determines the overall execution time of a parallel program.
- Explain how the activity demonstrates the concepts of speedup, scalability, and Amdahl's Law.

A complete description of the activity, including facilitator instructions, activity phases, discussion questions, timing recommendations, and assessment materials, is provided in the corresponding activity section 1.3.2.

8.3.2.4 Implementation Details

The Unplugged Penny Activity was performed in the Spring 2026 semester Table 8.6 summarizes the estimated time required for each component of the activity, from the introduction and setup through execution, discussion, and optional assessments. These estimates serve as a general guideline and may vary depending on class size, student engagement, and the amount of discussion incorporated.

Table 8.6: Estimated Instructional Time for Unplugged Penny Activity

Component	**Time (minutes)**
Pre-survey	3–5
Introduction and Instructions	5–10
Activity Setup	3–5
Activity Execution	25–30
Discussion and Debrief	6–10
Post-survey/Engagement Survey	3–5
Total Time	**45–65**

Students typically work in groups of five or six, though the instructor may accommodate smaller groups with minor activity modifications. Throughout the activity, students act as processors responsible for searching their assigned collection of pennies to identify the oldest coin. The instructor (or teaching assistant) manages task scheduling and distributes pennies

for each phase to illustrate different workload distributions. The activity consists of three progressively more complex execution scenarios:

Phase 1 – Sequential (Uniprocessor) Execution: One student performs the search, another records the execution time, and the remaining group members observe the process. This phase establishes a sequential performance baseline.

Phase 2 – Balanced Two-Processor Parallel Execution: The pennies are evenly divided between two students, resulting in a load-balanced parallel execution. Both students search simultaneously, communicate their results, and determine the overall oldest penny while another student records the total execution time. This phase demonstrates the benefits of parallel execution and the communication required between processors.

Phase 3 – Load-Imbalanced Five-Processor Parallel Execution: The pennies are distributed unevenly among five students, with one student receiving substantially more pennies than the others. This intentionally creates a load imbalance to illustrate how uneven workload distribution limits parallel performance and to demonstrate the practical implications of Amdahl's Law. An additional student records the overall execution time while the processors communicate to determine the oldest penny across all bags.

Following each phase, students compare execution times, discuss the impact of workload distribution and communication overhead, and relate their observations to fundamental PDC concepts. Complete implementation instructions are provided in Section 1.3.2.

- **Materials Required** The Unplugged Penny Activity uses readily available materials, with pennies serving as the primary instructional resource. The class size and the number of participating groups determine the number of pennies required. In situations where sufficient pennies are unavailable, such as after the discontinuation of penny production, paper replicas with different mint years may be used as an effective substitute without altering the learning objectives or the implementation of the activity. Required materials include:
  - One bag of pennies for each group (approximately 50 pennies, or another even number appropriate for the class size) for Phases 1 and 3.
  - Two bags of pennies for each group, together containing the same total number of pennies used in Phase 1, for the balanced parallel execution phase.
  - A timer, stopwatch, or mobile phone timer for each group.
  - *Optional:* A Penny Activity worksheet for recording sorting times, search times, and observations during each phase.
- **Instructional material (Lectures, Videos, Assignments, etc.)**

8.3.2.5 Experience and Teaching Tips

The Unplugged Penny Activity provides an engaging and intuitive introduction to data parallelism, scheduling, load balancing, speedup, and Amdahl's Law by allowing students to experience how data can be physically partitioned among processors. Before each phase begins, give groups a few minutes to discuss how the work will be organized and ensure that each participant clearly understands their role. Although the instructor assigns the data distribution for each phase, encouraging students to predict which phase will finish fastest and explain why promotes active engagement and creates a foundation for the post-activity

discussion. One of the greatest strengths of this activity is the contrast between the three execution scenarios. The sequential phase establishes a baseline for measuring performance, while the balanced two-processor phase demonstrates how evenly distributing data can reduce execution time and improve speedup. In contrast, the five-processor phase intentionally introduces a load imbalance by assigning one participant significantly more pennies than the others. Students quickly observe that the overall completion time depends on the slowest processor rather than the average completion time, providing a concrete illustration of why balanced workloads are critical for efficient parallel execution. Encourage students to pay attention not only to the sorting process but also to the communication required after each participant identifies the oldest penny in their assigned data. The time required to compare results and determine the oldest penny provides an excellent opportunity to discuss communication overhead and why additional processors do not always yield proportional performance improvements. This naturally leads into discussions of speedup and Amdahl's Law, as students recognize that portions of the task, such as combining results, remain sequential and limit the maximum achievable performance gain. Conclude the activity by comparing the recorded execution times from all three phases and asking students to explain the observed differences. Guide the discussion toward concepts such as balanced versus unbalanced data partitioning, scheduling decisions, communication overhead, and scalability. Encourage students to consider how they might redesign the workload in the five-processor phase to achieve better load balancing and greater speedup. These reflections help students connect their observations to fundamental PDC principles and reinforce that effective parallel computing depends not only on increasing the number of processors but also on distributing work efficiently and minimizing coordination overhead.

8.3.2.6 Data and Analysis

Following the Unplugged Penny Activity and the subsequent Plugged-In Parallel Sort Penny Activity, students completed a single engagement survey to evaluate their perceptions of the overall learning experience. The single post-activity survey was intentionally administered to minimize survey fatigue and reduce the burden on students. As a result, because the survey was completed only after both instructional activities had been finished, students' responses likely reflected their combined experiences with the unplugged and plugged-in activities rather than their perceptions of either activity individually.

The student engagement survey was adapted from the ASPECT [70] and modified to align with the learning objectives of the PDC activity. Although the survey was based on the ASPECT instrument, it was not administered in its original form. In addition to the engagement questions, the survey included several demographic items to provide participant background information. This section does not report demographic findings because the small sample size could compromise student confidentiality and limit the reliability of subgroup comparisons. Excluding these results helps protect student identities and avoids drawing potentially misleading conclusions from insufficiently represented demographic groups. The survey consisted of quantitative and qualitative items. The quantitative questions were rated using a five-point Likert scale, while the qualitative questions solicited open-ended feedback from students. For analysis, the quantitative survey questions were grouped according to the primary ASPECT categories. In addition, one question assessed students'

prior knowledge of PDC, and one question asked students to rate the extent to which the plugged-in activity helped them understand PDC concepts. Two open-ended questions asked students to provide suggestions for improving the unplugged activity and the plugged-in activity. Figure 8.2 illustrates the survey questions grouped by category for the unplugged and plugged-in Penny Sorting activities.

Category and Survey Questions
Prior Knowledge
• My knowledge of Parallel and Distributed Computing (PDC) before the activities.
Value of Group Activity*
• Explaining the material to my group improved my understanding of it.
• Having the material explained to me by my group members improved my understanding of the material.
• Group discussion during the Penny activity contributed to my understanding of parallel computing fundamentals.
• I had fun during today's Penny activity.
• Overall, the other members of my group made valuable contributions during the Penny activity.
• I would prefer to take a class that includes this group activity over one that does not include this Penny group activity.
• I am confident in my understanding of the material presented during today's Penny activity.
• The Penny activity increased my understanding of the fundamentals of parallel computing.
• The Penny activity stimulated my interest in parallel computing.
Personal Effort*
• I made a valuable contribution to my group today.
• I was focused during today's Penny activity.
• I worked hard during today's Penny activity.
Instructor Contribution*
• The instructor's enthusiasm made me more interested in the Penny activity.
• The instructor put a good deal of effort into my learning for today's class.
• The instructor seemed prepared for the Penny activity.
• The instructor and/or teaching assistants were available to answer questions during the Penny activity.
Programming Activity Effectiveness
• The plugged-in coding activity helped me understand PDC concepts.
Open-Ended Feedback
• How could the unplugged activity be improved?
• How could the plugged-in coding activity be improved?

Note. The student engagement survey was adapted from the *Assessing Student Perspective of Engagement in Class Tool* (ASPECT) [1]. The original ASPECT categories—*Value of Group Activity*, *Personal Effort*, and *Instructor Contribution*—were retained to evaluate student engagement following the unplugged and plugged-in Penny Sorting activities. Several survey items were modified to reference the PDC activities and their learning objectives. The three primary ASPECT categories, identified by an asterisk, were retained for analysis. Additional items assessed students' prior knowledge of PDC, the effectiveness of the subsequent plugged-in coding activity, and qualitative feedback regarding both instructional activities. Because the survey was administered after students completed both activities, responses may reflect students' perceptions of the overall learning experience rather than either activity individually.

Figure 8.2: Penny Activities Aspect Categories

Table 8.7 summarizes the quantitative survey results grouped according to the adapted ASPECT categories. Because the activity was implemented in a small Computer Science I course, the survey responses represent a limited sample of six students (n = 6). Consequently, the results should be interpreted as descriptive of this cohort rather than generalized to a broader student population. Students reported very limited prior knowledge of PDC before participating in the activities (M = 1.00), indicating that the concepts introduced were largely new to the class. Among the adapted ASPECT categories, **Instructor Contribution** received the highest mean score (M = 4.71), suggesting that students viewed the in-

structor as well prepared, enthusiastic, and readily available to answer questions throughout the activities. The **Personal Effort** category also received a high rating (M = 4.5), indicating that students remained focused, worked diligently, and believed they made meaningful contributions to their groups during the activity. Students rated the **Value of the Group Activity** positively (M = 4.26), suggesting that the collaborative nature of the Penny Activities supported their understanding of foundational parallel computing concepts. Students generally agreed that explaining concepts to peers, receiving explanations from group members, participating in group discussions, and working collaboratively contributed to their learning while making the activity engaging and enjoyable. The **Programming Activity Effectiveness** also received favorable ratings (M = 4.00), indicating that students believed it reinforced the PDC concepts introduced during the unplugged exercise. Although this category received the lowest mean among the post-activity categories, the rating remained positive, suggesting that the programming activity effectively complemented the conceptual understanding developed during the unplugged activity. Student comments, however, indicate that additional time spent reviewing or writing the code could further strengthen this component of the lesson. While the small sample size limits broad conclusions, the findings suggest that combining an unplugged collaborative activity with a subsequent programming exercise can provide an engaging and effective introduction to foundational PDC concepts for students with little or no prior PDC experience. Analysis of the open-ended responses revealed several recurring themes. Students appreciated the collaborative structure of the unplugged activity and the opportunity to communicate and coordinate with their peers while completing the task. Several students suggested incorporating additional practice or trial runs before beginning the activity, while others noted that a larger class size would provide greater opportunities for interaction. Feedback regarding the plugged activity primarily focused on increasing the amount of instructional time devoted to the programming exercise, allowing students to write the parallel code themselves, providing a more detailed walkthrough of the implementation, and incorporating additional visual demonstrations. One student also noted that an instructional video failed to load during the activity. Overall, the qualitative feedback complements the quantitative findings by suggesting that students valued both instructional components while identifying opportunities to strengthen the programming portion of the lesson further.

Table 8.7: Student Engagement Survey Results Grouped by Adapted ASPECT Categories for the Penny Activities, $n = 6$

Category	**Items**	**Mean**
Prior Knowledge	1	1.00
Value of the Group Activity (ASPECT)	9	4.26
Personal Effort (ASPECT)	3	4.50
Instructor Contribution (ASPECT)	4	4.71
Plugged-in Activity Effectiveness	1	4.00
Overall	**18**	**4.20**

8.4 New Plugged-in Activities

8.4.1 Plugged-in - Parallel Sort Penny Activity

8.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

The Plugged-In Parallel Sort Penny Activity builds on the Unplugged Penny Activity by introducing students to the programmatic implementation of sequential and parallel sorting in Java. The activity is designed to help students connect the parallel computing concepts experienced during the hands-on activity with real-world programming techniques. Because many undergraduate students have little or no prior exposure to parallel programming, the activity provides an accessible introduction to how modern programming languages and libraries support parallel execution. Students examine and compare sequential and parallel implementations while exploring concepts such as task decomposition, concurrency, synchronization, load balancing, scalability, and speedup. Two Java programs support the activity. The first establishes a sequential baseline for comparison, while the second demonstrates multiple approaches to parallel programming using Java's built-in parallel libraries and the Fork/Join framework. By comparing these implementations, students gain an understanding of the different levels of abstraction available for parallel programming and the trade-offs between implementation simplicity, programmer control, and performance. The activity reinforces the relationship between the unplugged exercise and practical software development by illustrating how the same parallel-computing concepts are applied in real programs.

8.4.1.2 Prerequisites

Students should have:

- Completed the Unplugged Penny Activity.
- A basic understanding of Java programming, including variables, loops, methods, arrays, and classes.
- An understanding of sequential program execution.
- Experience reading and interpreting Java source code.
- No prior experience with parallel programming is required.

8.4.1.3 Learning Outcomes

After completing this activity, students should be able to:

- Explain the differences between sequential and parallel execution.
- Describe the principles of data, functional, and task parallelism.
- Explain how Java supports parallel programming through built-in libraries and the Fork/Join framework.
- Describe the roles of task decomposition, concurrency, synchronization, load balancing, communication overhead, and speedup in parallel applications.
- Explain how different parallel programming approaches vary in implementation complexity, programmer control, scalability, and performance.

- Explain the connection between the Penny Sorting activity and its corresponding implementation in Java programs.

8.4.1.4 Implementation Details

Table 8.8 summarizes the estimated time required for each component of the activity, from the introduction and setup through execution, discussion, and optional assessments. These estimates serve as a general guideline and may vary depending on class size, student engagement, and the amount of discussion incorporated.

Table 8.8: Estimated Instructional Time for Plugged-In Parallel Sort Penny Activity

Component	**Time (minutes)**
Pre-survey	3–5
Introduction and Instructions	2–5
Activity Setup	2–5
Activity Execution	15–20
Discussion and Debrief	5–10
Post-survey/Engagement Survey	3–5
Total Time	**30–50**

The Plugged-In Parallel Sort Penny Activity is designed as a guided code review and demonstration that connects the concepts introduced in the Unplugged Penny Activity to Java implementations. Rather than requiring students to develop parallel programs, the activity helps them recognize how fundamental PDC concepts are represented in real code. Students examine and execute one sequential and three parallel sorting programs, compare their behavior and performance, and discuss the tradeoffs among different approaches to parallel programming. Throughout the activity, the instructor emphasizes high-level concepts such as task decomposition, data partitioning, concurrent execution, coordination, and speedup, rather than the underlying algorithms or runtime implementation details.

The activity is implemented through the following five instructional steps:

1. **Introduce the Sequential Program.** Review the sequential Java sorting program with the class. Explain the major components of the program, demonstrate how it sorts the data using `Arrays.sort()`, and have students run the program several times to establish a sequential baseline.
2. **Present the Parallel Implementations.** Introduce three parallel versions of the sorting program that use `Arrays.parallelSort()`, the Java Stream API, and the Fork/Join framework. Provide a high-level explanation of how each implementation divides work and executes tasks concurrently while emphasizing the similarities and differences among the approaches.
3. **Guide Students Through Program Execution.** Have students compile and run each implementation using the same input data. Students record execution times, observe how each approach performs the sorting task, and compare the results with the sequential baseline.
4. **Facilitate Comparison and Discussion.** Lead a discussion comparing the sequential and parallel implementations. Encourage students to consider differences in ex-

ecution time, programming complexity, task organization, and the tradeoffs between ease of use, programmer control, and parallel overhead.

5. **Connect to PDC Concepts.** Conclude by relating the Java implementations back to the unplugged Penny Sorting activity. Reinforce how concepts such as task decomposition, data partitioning, concurrent execution, coordination, synchronization, and speedup are represented in working Java programs, providing a foundation for more advanced study of parallel programming.
   By progressing through these steps, students develop a high-level conceptual understanding of how parallel programming extends familiar sequential programming concepts. Instead of emphasizing low-level implementation details, the activity focuses on the core principles of parallel execution while reinforcing the connections between the unplugged activity and Java implementations.

- **Materials Required** The Plugged-In Parallel Sort Penny Activity requires minimal instructional resources and is designed to be delivered primarily as an instructor-led demonstration. The following materials are required to implement the activity:
  - An instructor computer with the Java Development Kit (JDK) and a Java Integrated Development Environment (IDE), such as IntelliJ IDEA, Eclipse, or NetBeans.
  - The provided sequential and parallel Java programs used to demonstrate the PDC concepts.
  - A classroom projector or large display for presenting the source code, program execution, and results to students.
  - *Optional:* Presentation slides to introduce the concepts and summarize the results.
  - *Optional:* Student handouts or worksheets for recording observations and comparing the sequential and parallel implementations.
- **Instructional material (Lectures, Videos, Assignments, etc.)**

8.4.1.5 Experience and Teaching Tips

The transition from the Unplugged Penny Activity to the Plugged-In Parallel Sort Penny Activity provided students with an opportunity to connect their conceptual understanding of PDC with practical programming techniques. Before introducing the code, it was helpful to revisit the unplugged activity and ask students to describe how the work had been divided among participants, how communication occurred, and what factors influenced the overall execution time. This discussion allowed students to recognize that the Java programs implemented the same ideas they had already experienced in the hands-on activity.

Because the students had little or no prior exposure to parallel programming, reviewing the code line by line was more effective than expecting students to immediately understand the complete programs independently. Comparing the sequential implementation with the three parallel implementations allowed students to observe that there are multiple ways to express parallelism in Java, each with different levels of abstraction, complexity, and programmer control. Emphasizing the similarities between the sequential and parallel versions before discussing their differences helped reduce student anxiety and kept the focus on the

underlying PDC concepts rather than language syntax.

One lesson learned from the initial implementation was that combining both the unplugged and plugged activities into a single class session created a substantial cognitive load for students. Although students found the coding examples valuable, many indicated that they would have benefited from additional time to review the programs, step through the execution, and modify portions of the code themselves. Future offerings would benefit from presenting the coding activity during a separate class session after students have had time to reflect on the unplugged exercise. This approach would provide more opportunities for guided practice, discussion, and experimentation while reinforcing the connection between conceptual models and practical parallel programming.

Finally, instructors should encourage students to analyze the benchmark results rather than focusing solely on execution times. The discussion should emphasize why multiple executions are averaged, how JVM warm-up and system variability affect performance measurements, and why increased parallelism does not always result in proportional speedup. Connecting these observations back to the Unplugged Penny Activity reinforces that the same principles of data partitioning, load balancing, communication overhead, and scalability apply whether the processors are students sorting pennies or threads executing code.

8.4.1.6 Data and Analysis

As discussed in Section 8.3.2.6, students completed a single engagement survey after participating in both the unplugged and plugged-in activities. Because the survey was administered after both instructional components had been completed, students' responses likely reflected their overall perceptions of the combined learning experience rather than either activity in isolation. For this reason, a more detailed analysis of the engagement survey is presented in Unplugged Penny Activity Data and Analysis section 8.3.2.6. As shown in Figure 8.2 the survey included one item specifically designed to evaluate the effectiveness of the plugged-in activity. Students responded on a 5-point Likert scale ranging from 1 (*Strongly Disagree*) to 5 (*Strongly Agree*). The survey included the following item:

> *"The plugged-in coding activity helped me understand PDC concepts."*

Overall, students responded positively, indicating that they perceived the activity as contributing to their understanding of parallel and distributed computing concepts, with a mean rating of M = 4.00. These results suggest that the Plugged-In Parallel Sort Penny Activity may have reinforced concepts introduced through the unplugged exercises by letting students connect the underlying ideas to their Java implementation.

Qualitative feedback further supported this finding while identifying opportunities for improvement. Several students recommended allocating additional class time for the programming exercise, providing more detailed walkthroughs of the implementation, and allowing students to write portions of the parallel code themselves. One student also suggested incorporating additional visual demonstrations to complement the coding examples. Given the small sample size (n = 6), these findings should be interpreted as descriptive of this cohort rather than generalized to a broader population. Future studies involving larger cohorts, multiple course offerings, and separate evaluations of the unplugged and plugged-in

activities would help validate these findings and provide a more detailed assessment of the instructional effectiveness of each component individually as well as the combined instructional approach.

8.5 Course-wide Pre and Post Course Surveys

The pre- and post-course surveys collected demographic information, asked students to self-assess their overall computer experience, and measured their perceived experience with various programming languages and programming concepts. Students rated each experience item on a five-point Likert scale, where 1 indicated no experience and 5 indicated extensive experience. Although demographic information was collected, it was not included in the analyses presented in this chapter due to the small sample size and the potential risk of identifying individual students. Instead, the analysis focuses on students' self-reported experience with Java, the programming language used in the course, along with the following programming topics: Data Types, Arithmetic Expressions and Operators, Branching, Loops, Calling Functions/Methods, Arrays, Writing Static Functions, Writing Methods, Creating Objects, Parallel Processing, Distributed Computing, and Event Handling. Comparing students' pre- and post-survey responses provides insight into changes in their perceived knowledge and confidence throughout the course. The comparison tables presented in this section report the mean pre- and post-survey ratings for each programming topic, along with the corresponding mean change between the two surveys. This section examines the survey results across three course offerings. The Fall 2024 offering served as a baseline because no PDC instructional activities were incorporated into the course. In contrast, the Fall 2025 and Summer 2026 offerings integrated PDC activities into the curriculum, allowing changes in students' confidence in both traditional programming topics and introductory PDC concepts to be examined.

The Fall 2024 pre- and post-course surveys were used to evaluate changes in students' self-reported programming knowledge and confidence over the semester. Five students completed both surveys and were included in the comparison. Table 8.9 summarizes students' self-assessed experience before and after the course across thirteen programming and computing topics using a five-point Likert scale, where higher ratings indicate greater perceived knowledge and confidence. Overall, students reported increased confidence across nearly all programming topics following instruction, suggesting that the course successfully strengthened their understanding of fundamental programming concepts. The largest gains were observed in Writing Methods and Creating Objects, each increasing by 2.0 points on the five-point scale. Substantial improvements were also observed in Writing Static Functions (+1.8), Calling Functions/Methods (+1.2), Arrays (+1.2), and Arithmetic Expressions and Operators (+1.0), indicating increased confidence in many of the core programming skills emphasized throughout the semester. Moderate gains were observed in Java, Branching, and Loops (each +0.8), while Data Types increased by 0.4 points. The variation in the reported gains across topics may also have been influenced by the timing of the post-course survey, which was administered at the end of the semester. Topics such as Writing Methods, Creating Objects, and Writing Static Functions were covered later in the course and therefore may have been more recent in students' minds when completing the survey. In contrast,

foundational topics such as Data Types, Branching, and Loops were introduced much earlier in the semester. Consequently, differences in the magnitude of the reported gains may reflect not only students' perceived learning but also the recency of instruction. Students reported only modest increases in confidence related to Parallel Processing, which increased from 1.2 to 1.8 (+0.6). Distributed Computing showed no overall change, while Event Handling increased only slightly (+0.2). These findings are consistent with the course design, as no dedicated PDC instructional activities were incorporated into this offering. As a result, students had limited opportunities to develop confidence in PDC-related concepts, and their familiarity with these topics remained comparatively low. Given the small sample size ($n = 5$), these findings should be interpreted as descriptive rather than inferential. Overall, the results suggest that students perceived meaningful growth in traditional programming knowledge and object-oriented programming skills throughout the course while providing a useful baseline for comparison with subsequent course offerings that incorporated dedicated PDC instruction.

Table 8.9: Comparison of Pre- and Post-Course Survey Results (Fall 2024, $n = 5$)

Experience Category	**Pre**	**Post**	**Change**
Java	2.40	3.20	+0.80
Data Types	3.40	3.80	+0.40
Arithmetic Expressions	3.00	4.00	+1.00
Branching	3.00	3.80	+0.80
Loops	3.00	3.80	+0.80
Calling Functions/Methods	3.00	4.20	+1.20
Arrays	2.40	3.60	+1.20
Writing Static Functions	2.20	4.00	+1.80
Writing Methods	2.20	4.20	+2.00
Creating Objects	2.20	4.20	+2.00
Parallel Processing	1.20	1.80	+0.60
Distributed Computing	1.80	1.80	0.00
Event Handling	2.00	2.20	+0.20

The Fall 2025 and Spring 2026 CS1 courses included specific PDC activities. The Fall 2025 pre- and post-course surveys were used to evaluate changes in students' self-reported programming knowledge and confidence over the semester. Four students completed both surveys and were included in the comparison. Table 8.10 summarizes students' self-assessed confidence before and after instruction across thirteen programming and computing topics using a five-point Likert scale, where higher ratings indicate greater perceived knowledge and confidence. Overall, students reported increased confidence across all topics after completing the course. The largest improvement was observed in students' self-rated Java proficiency, with the mean increasing from 1.50 to 3.50 (+2.00). This result is expected, as Java programming served as the primary focus of the CS1 course and provided the foundation for introducing object-oriented programming concepts throughout the semester. Other substantial gains were observed in Writing Static Functions (+1.75), Arrays (+1.50), Parallel Processing (+1.50), and Event Handling (+1.50). Moderate improvements were found in Writing Methods, Creating Objects, and Distributed Computing (each +1.25), while Calling Functions/Methods increased by 1.00 point. Smaller gains were observed in Data Types, Branching, and Loops (each +0.75), and Arithmetic Expressions and Operators showed the

smallest increase (+0.50). Unlike the Fall 2024 course offering, the Fall 2025 course incorporated introductory PDC concepts through an unplugged learning activity. Students reported meaningful increases in confidence for both Parallel Processing (+1.50) and Distributed Computing (+1.25), suggesting that even a relatively brief introduction to PDC concepts can improve students' familiarity with these topics. Although students' post-course confidence in PDC remained somewhat lower than that reported for many traditional programming concepts, the observed gains indicate that the integrated activities successfully introduced fundamental PDC concepts without detracting from students' development of core programming skills. As with the Fall 2024 results, the post-course survey was administered at the end of the semester. Consequently, topics covered later in the course may have been more recent in students' minds when completing the survey than concepts introduced earlier, and some differences in the magnitude of the reported gains may reflect both students' perceived learning and the recency of instruction. Because the sample size was small ($n = 4$), these findings should be interpreted as descriptive rather than inferential. Nevertheless, the overall pattern suggests meaningful growth in students' confidence across traditional programming topics while also demonstrating substantially greater gains in PDC-related concepts than were observed in the Fall 2024 course, providing preliminary evidence that integrating introductory PDC activities into CS1 can increase students' familiarity with parallel and distributed computing concepts.

Table 8.10: Comparison of Pre- and Post-Course Survey Results (Fall 2025, $N = 4$)

Experience Category	**Pre**	**Post**	**Change**
Java	**1.50**	**3.50**	**+2.00**
Data Types	3.25	4.00	+0.75
Arithmetic Expressions & Operators	3.25	3.75	+0.50
Branching	3.00	3.75	+0.75
Loops	2.75	3.50	+0.75
Calling Functions/Methods	2.75	3.75	+1.00
Arrays	2.00	3.50	+1.50
Writing Static Functions	1.75	3.50	+1.75
Writing Methods	2.00	3.25	+1.25
Creating Objects	2.25	3.50	+1.25
Parallel Processing	1.50	3.00	+1.50
Distributed Computing	1.75	3.00	+1.25
Event Handling	1.75	3.25	+1.50

The Spring 2026 pre- and post-course surveys were used to evaluate changes in students' self-reported programming knowledge and confidence over the semester. Five students completed both surveys and were included in the comparison. Table 8.11 summarizes students' self-assessed confidence before and after instruction across thirteen programming and computing topics using a five-point Likert scale, where higher ratings indicate greater perceived knowledge and confidence. Building on the Fall 2025 offering, Spring 2026 was the first semester to incorporate a complementary plugged-in activity alongside the unplugged PDC activity, providing students with an opportunity to connect the concepts introduced during the hands-on exercises to parallel programming techniques implemented in Java.

Overall, students reported increased confidence across all programming and computing topics following instruction. The largest improvement was observed in Writing Static Func-

tions, with the mean increasing from 1.00 to 4.00 (+3.00). This substantial gain suggests that students developed considerably greater confidence in designing and implementing reusable program components during the course. Other notable improvements were observed in Writing Methods (+2.80), as well as Java, Calling Functions/Methods, and Arrays (each +2.60). Additional substantial gains were observed in Branching and Creating Objects (each +2.40), reflecting increased confidence in both fundamental programming constructs and object-oriented programming concepts. Moderate improvements were observed in Data Types and Loops (each +1.80), while Arithmetic Expressions and Operators increased by 1.60 points.

Students also reported increased confidence in Parallel Processing, Distributed Computing, and Event Handling, each improving by 1.00 point. Although these gains were smaller than those observed for many traditional programming topics, the post-course ratings indicate greater familiarity and confidence following students' initial exposure to PDC concepts. Because students began the course with relatively little experience in these areas, the observed improvements suggest that the integrated unplugged and plugged-in activities successfully introduced foundational PDC concepts while reinforcing students' understanding of how these concepts can be applied in Java programs without detracting from the development of core programming skills. As with the Fall 2024 and Fall 2025 offerings, the post-course survey was administered at the end of the semester. Consequently, topics covered later in the course may have been more recent in students' minds when completing the survey than concepts introduced earlier, and some variation in the reported gains may reflect both students' perceived learning and the recency of instruction. Given the small sample size ($n = 5$), these findings should be interpreted as descriptive rather than inferential. Nevertheless, the overall pattern suggests meaningful growth in students' confidence across traditional programming topics while also demonstrating increased familiarity with introductory PDC concepts following the integrated instructional activities.

Table 8.11: Comparison of Pre- and Post-Course Survey Results (Spring 2026, $n = 5$)

Experience Category	**Pre**	**Post**	**Change**
Java	**1.20**	**3.80**	**+2.60**
Data Types	2.20	4.00	+1.80
Arithmetic Expressions & Operators	2.40	4.00	+1.60
Branching	1.60	4.00	+2.40
Loops	2.20	4.00	+1.80
Calling Functions/Methods	**1.40**	**4.00**	**+2.60**
Arrays	**1.20**	**3.80**	**+2.60**
Writing Static Functions	**1.00**	**4.00**	**+3.00**
Writing Methods	**1.20**	**4.00**	**+2.80**
Creating Objects	1.60	4.00	+2.40
Parallel Processing	1.80	2.80	+1.00
Distributed Computing	1.80	2.80	+1.00
Event Handling	2.00	3.00	+1.00

8.5.0.1 Course-wide Pre and Post Course Surveys Overall Conclusions

Across the three course offerings, students consistently reported increased confidence in traditional programming topics, including Java, data types, arithmetic expressions, branch-

ing, loops, arrays, calling functions and methods, writing static functions, writing methods, and creating objects. The largest gains were consistently observed in topics related to procedural and object-oriented programming, particularly writing functions and methods, manipulating arrays, and creating objects. These are fundamental CS1 learning outcomes that provide the foundation for later coursework in software development and computer science. The observed improvements suggest that students perceived meaningful growth in both their programming knowledge and their ability to apply core programming concepts throughout the semester.

An important observation across the three course offerings is that the introduction of PDC activities did not appear to detract from students' perceived learning of the traditional CS1 programming topics. Students in the Fall 2025 and Spring 2026 offerings, which incorporated PDC activities, reported improvements across the same foundational programming concepts as students in the baseline Fall 2024 offering. In many cases, the gains in Java programming, arrays, functions, methods, and object-oriented programming concepts were comparable to or greater than those observed in the course offering that did not include PDC activities. These findings indicate that incorporating PDC instruction did not come at the expense of the core CS1 curriculum. Instead, students continued to report substantial gains in the traditional programming concepts while also developing familiarity with introductory PDC topics. Comparing the baseline course offering (Fall 2024) with the two offerings that incorporated PDC activities reveals a notable difference in students' self-reported confidence related to Parallel Processing and Distributed Computing. During the Fall 2024 offering, when no dedicated PDC instructional activities were included, students reported only modest improvement in Parallel Processing, no improvement in Distributed Computing, and minimal gains in Event Handling. In contrast, students in both the Fall 2025 and Spring 2026 offerings reported larger increases in confidence for Parallel Processing and Distributed Computing after participating in the unplugged and plugged-in PDC activities. Although post-course ratings for these topics remained lower than those for traditional programming concepts, the observed improvements suggest that students completed the courses with greater familiarity and confidence in these topics than they reported at the beginning of the semester.

The Spring 2026 course generally exhibited the largest increases in self-reported confidence across many of the traditional programming topics. Students reported substantial gains in Java, branching, arrays, writing static functions, writing methods, creating objects, and calling functions, while also demonstrating positive increases in Parallel Processing and Distributed Computing. The progression of results across the three course offerings suggests that students can develop confidence in advanced computing topics while simultaneously strengthening their understanding of fundamental programming concepts introduced throughout the CS1 curriculum. Student engagement survey responses complemented these findings. Students generally agreed that the programming activity reinforced the concepts introduced during the unplugged activity and helped them better understand parallel computing. Qualitative feedback indicated that students appreciated the connection between the physical activities and their software implementations. Students also suggested several improvements, including allocating additional time for coding, incorporating more instructor-led demonstrations, allowing students to develop portions of the code themselves, and providing additional visual explanations of the algorithms. These recommendations provide valuable guidance for refining future implementations while maintaining the connection

between conceptual understanding and practical programming.

Because these findings are based on self-reported confidence rather than direct measures of programming performance, they reflect students' perceptions of their learning. They should be interpreted alongside the activity assessments, engagement survey results, and other evaluation measures presented in this chapter. Additionally, limited enrollment across course offerings restricts how far these findings can be generalized beyond the students included in the study. Although statistical conclusions cannot be drawn from these data, the consistent patterns observed across multiple course offerings provide encouraging preliminary evidence regarding students' perceived growth in traditional programming knowledge and their increased familiarity with foundational PDC concepts following the instructional activities. Future studies involving larger cohorts, multiple institutions, longitudinal assessment, and multiple assessment methods, including both direct and indirect measures of learning, will provide a more comprehensive evaluation of the effectiveness of these instructional approaches and their impact on student learning.

8.6 Other Activities and Ideas

8.6.1 Unplugged Activity - More Processors (MP) Writing Activity

8.6.1.1 Problem Description, Prerequisites, and Learning Outcomes

The MP Writing Activity is an unplugged classroom activity that introduces students to fundamental parallel computing concepts through a collaborative, hands-on simulation. The activity requires only paper, pencils or pens, and a timer, making it easy to implement in most classroom settings. Students assume the role of processors and complete a series of writing tasks, with execution times recorded. As the activity progresses, students experience how different execution models affect performance and develop an intuitive understanding of sequential execution, concurrency, shared-memory parallelism, scheduling, load balancing, speedup, scalability, and the limitations described by Amdahl's Law.

8.6.1.2 Prerequisites

There are no prerequisites for this activity, making it suitable for students with little or no prior exposure to programming or PDC concepts.

8.6.1.3 Learning Outcomes

Upon completion of the activity, students should be able to:

- Explain the differences between sequential execution and parallel execution.
- Explain how partitioning data across multiple processors supports parallel processing.
- Describe how distributing work among processors contributes to effective load balancing.
- Identify the communication and synchronization overhead that can occur during parallel execution.

- Explain why the processor that finishes last determines the overall execution time of a parallel program.
- Explain how the activity demonstrates the concepts of speedup, scalability, and Amdahl's Law.

8.6.1.4 Implementation Details

Table 8.12 summarizes the estimated time required for each component of the activity, from the introduction and setup through execution, discussion, and optional assessments. These estimates serve as a general guideline and may vary depending on class size, student engagement, and the amount of discussion incorporated.

Table 8.12: Estimated Instructional Time for Unplugged MP Writing Activity

Component	**Time (minutes)**
Pre-survey	3–5
Introduction and Instructions	5–10
Activity Setup	3–5
Activity Execution	20–25
Discussion and Debrief	5–10
Post-survey/Engagement Survey	3–5
Total Time	**39–60**

The activity is divided into five phases:

Phase 1 – Sequential (Uniprocessor) Execution: One student sequentially writes the sentence *"More Processors Are Not Always The Best"* and then assigns an index number to each character, while a second student records the execution time. This phase establishes a baseline for sequential execution.

Phase 2 – Single-Processor Multitasking Execution: One student alternates between writing each character of the sentence and immediately assigning its corresponding index, while a second student records the execution time. This phase demonstrates multitasking on a single processor and allows students to compare interleaved execution with purely sequential execution.

Phase 3 – Two-Processor Parallel Execution: One student writes the sentence while a second student simultaneously assigns character indices. A third student records the execution time, with the overall completion time determined by the slower processor. This phase demonstrates parallel execution while introducing communication and synchronization between processors.

Phase 4 – Four-Processor Parallel Execution: Four students divide the writing and indexing tasks into separate portions of the sentence while a fifth student records the execution time. Communication is required to transfer the next available character index before the remaining portion of the sentence can be indexed, illustrating task decomposition, workload distribution, inter-process communication, synchronization, and load balancing.

Optional Phase 5 – An additional task, counting the number of characters in each word, is introduced while increasing both the workload and the number of processors. This phase

demonstrates weak scaling by increasing the problem size with the available processing resources, while also illustrating that sequential portions of a problem ultimately limit the achievable speedup, as described by Amdahl's Law.

Following each phase, students compare execution times and discuss how the execution model influenced performance. Guided reflection encourages students to examine the effects of task decomposition, processor count, workload distribution, communication, synchronization, concurrency, scalability, and load balancing, while identifying which portions of the activity could and could not be parallelized. By physically performing the tasks and observing the timing differences firsthand, students develop an intuitive understanding of fundamental PDC concepts before implementing similar algorithms in code.

- **Materials Required**
  - Blank paper for each student group to complete the writing and indexing tasks.
  - Writing utensils (e.g., pens or pencils).
  - A timer, stopwatch, smartphone timer, or other digital timing device to measure the execution time for each phase.
- **Instructional material (Lectures, Videos, Assignments, etc.)**

8.6.1.5 Experience and Teaching Tips

The activity requires only paper, writing utensils, and a timer, making it a low-cost and easily deployable exercise that can be incorporated into nearly any classroom setting with minimal preparation. Because no specialized hardware or software is needed, instructors can easily incorporate the exercise into a standard class period with minimal preparation. Experience teaching this activity: provide clear instructions before the activity begins, assign specific roles (e.g., processors and timer), and allow students to rotate through different roles to foster greater engagement and a deeper understanding of the underlying concepts. It is also helpful to conclude the activity with a guided discussion that encourages students to reflect on how communication overhead, synchronization, and task dependencies affected performance and to connect these observations to parallel computing principles such as scalability and Amdahl's Law.

8.6.2 Ideas on Activities

The Unplugged Flag Maker Activity is simple to implement but requires advance preparation and coordination. Instructors must prepare gridded flag templates, colored markers or crayons, and timers for each group. Students may also need time to understand their assigned processor roles and coordinate their work, particularly during the parallel phases where they share the same worksheet and drawing tools. These shared resources intentionally create contention and synchronization challenges, providing natural opportunities to discuss mutual exclusion, scheduling, and communication in parallel systems. Several alternatives can simplify implementation while maintaining the learning objectives. Instructors may substitute the Mauritius flag for another gridded image, pixel art, or a simple geometric design; use colored pencils, stickers, or other coloring materials; or adapt the activity for online or hybrid courses using collaborative drawing tools. The activity can also be scaled by adjust-

ing the grid size, number of colors, or number of processors to accommodate different class sizes and instructional goals.

The primary logistical challenge for the Unplugged Penny Activity is obtaining, transporting, and organizing enough pennies for the class. Depending on class size, several hundred or even thousands of pennies may be required. Instructors should plan ahead to acquire sufficient pennies and store them in labeled bags that are already divided for each phase of the activity. As an alternative, laminated or paper-cut pennies with printed years can be used to reduce cost, weight, and preparation time while still preserving the activity's learning objectives. These cutouts are easier to transport, organize, and redistribute between phases, making them especially useful for large classes or conference workshops.

The Unplugged MP Writing Activity is an easy-to-implement unplugged activity that requires only paper, pencils or pens, printed instructions, and a timer. The Unplugged MP Writing Activity is flexible and can be adapted for a variety of classroom settings. Instructors may substitute the original sentence with another sentence, code fragment, or mathematical expression, add additional tasks to illustrate concepts such as task decomposition or weak scaling, or adjust the number of participants to accommodate different class sizes. The activity can also be adapted for online or hybrid instruction using collaborative editing tools. To maximize student learning, instructors should encourage students to predict the performance of each execution model before beginning a phase and then compare those predictions with the recorded results. These discussions help students connect the hands-on experience to fundamental parallel computing principles and reinforce the relationship between theoretical concepts and practical execution.

The Plugged-In Parallel Sort Penny Activity is easy to implement because it uses standard Java libraries and a provided Fork/Join implementation. Students begin by reviewing and running a sequential sorting program before exploring three parallel implementations, helping them connect the concepts introduced in the Unplugged Penny Activity with real Java code. Since the activity is designed for a CS1 course, instruction should emphasize high-level concepts such as data decomposition, task decomposition, concurrency, coordination, and parallel overhead rather than the internal details of Java's parallel libraries or sorting algorithms. Instructors may extend the activity by experimenting with different array sizes, having students make minor modifications to the provided programs, or comparing execution times on different hardware platforms while maintaining the focus on foundational PDC concepts.

8.7 Conclusion

Introducing PDC concepts early in the computer science curriculum helps students develop a strong conceptual foundation for topics they will encounter throughout their academic and professional careers. As described in this chapter, the instructor integrated several unplugged activities and one plugged-in activity into an introductory CS1 course to introduce foundational PDC concepts without requiring prior experience in parallel programming. Through hands-on, collaborative learning experiences, students explored concepts such as concurrency, message passing, shared memory, scheduling, load balancing, speedup, and Amdahl's Law in an engaging and accessible way. The unplugged activities made "paral-

lel thinking" approachable by allowing students to physically simulate parallel processes, while the plugged activity reinforced these concepts by demonstrating their application in a computational setting. The unplugged activities primarily addressed the Remembering and Understanding levels of Bloom's Revised Taxonomy, whereas the plugged activity provided opportunities for students to Apply these concepts in a computational setting. Collectively, these activities established a conceptual foundation upon which students can later develop higher-order skills associated with Analyzing, Evaluating, and Creating as they progress through more advanced PDC coursework.

Several lessons emerged from this work. First, introducing PDC concepts through active, collaborative learning can make topics that are often perceived as abstract or advanced more accessible to beginning programmers. Students responded positively to the unplugged activities, which encouraged discussion, teamwork, and problem-solving before introducing implementation details. Second, combining unplugged and plugged activities provided a natural progression from conceptual understanding to computational application, helping students connect theoretical ideas with programming practice. Finally, embedding PDC concepts within existing CS1 topics demonstrated that meaningful PDC instruction can be integrated into an introductory programming course without requiring a separate course or substantial curricular changes.

Based on these experiences, several teaching recommendations may benefit instructors interested in incorporating PDC into early computing courses. Instructors should clearly connect each activity to familiar CS1 concepts, such as loops, arrays, or methods, so students recognize the relevance of PDC within the broader curriculum. Sufficient time should be allocated for discussion and debriefing after each activity, as these conversations often help students relate their experiences to formal PDC terminology. Small-group collaboration should be encouraged to promote peer learning, while instructors should actively facilitate discussions and provide guidance as needed. Finally, unplugged activities should be paired with a simple programming exercise whenever possible to reinforce conceptual understanding through practical application.

Although the results are encouraging, several limitations should be considered. The study was conducted at a single institution with relatively small class sizes, limiting the generalizability of the findings. Much of the evaluation relied on student perceptions, self-reported surveys, and short learning assessments rather than long-term measures of knowledge retention or programming performance. Participation also varied across the assessment instruments. Not all students completed both the course pre- and post-surveys or the activity-specific surveys associated with the unplugged and plugged-in activities. Consequently, the findings reported for the different assessments are based on the available responses for each instrument and should be interpreted accordingly. Furthermore, the activities were implemented within one CS1 course; therefore, the findings may not directly transfer to institutions with different student populations, programming languages, or curricular structures.

Future work will focus on expanding both the implementation and evaluation of these instructional activities. Additional unplugged and plugged-in activities will be developed to introduce a broader range of foundational PDC concepts and provide students with increasingly rich opportunities to connect conceptual understanding with computational practice. Multi-institutional studies involving larger and more diverse student populations, together with longitudinal data collection, will provide stronger evidence regarding the effectiveness

of integrating PDC concepts early in the computing curriculum. Future research will also examine how early exposure to PDC influences student performance and preparedness in subsequent computer science courses, particularly upper-level PDC, systems, software engineering, database systems, and other courses where concurrency and parallelism play an important role.

Continued refinement of the instructional materials based on student input and classroom experiences will further improve their effectiveness and usability. Finally, ongoing assessment using validated learning instruments, performance-based measures, and long-term follow-up studies will provide deeper insight into how early PDC instruction influences students' conceptual understanding, programming ability, problem-solving skills, and long-term retention of fundamental parallel and distributed computing concepts.

Appendix

Github Link to Instructional Material

The HPU CS1 GitHub repository, which contains the PDC intervention materials described in this chapter, is available at: *HPU-CS1-PDC-Github*.

Detailed Pre and Post Syllabus

A PDC intervention was implemented in CSCI 2911/2916 Computer Science I (CS1) at Hawai'i Pacific University. The CS1 course consists of CSCI 2911, a three-credit lecture, and CSCI 2916, a one-credit laboratory, and is taught using Java with Gaddis's Starting Out with Java: Control Structures through Objects (Pearson Revel Edition). The overall organization and progression of the course remained largely unchanged following the integration of PDC concepts, ensuring that the existing CS1 curriculum was preserved. In both the pre- and post-intervention syllabi, students progress through the same sequence of core programming topics, including Java fundamentals, decision structures, loops and files, methods, text processing and regular expressions, classes, arrays and ArrayLists, unit testing with JUnit, and introductory graphical user interface (GUI) development before concluding with a course wrap-up. The course continued to emphasize foundational programming skills, problem solving, and object-oriented programming principles. The primary curricular enhancement in the post-intervention syllabus was the addition of a dedicated PDC Fundamentals unit near the end of the semester. This placement allowed students to first develop sufficient proficiency with Java programming, classes, arrays, and algorithmic problem solving before being introduced to fundamental concepts in PDC. The revised syllabus also incorporated PDC-related lectures, discussions, and unplugged and plugged-in activities within this unit while maintaining full coverage of the traditional CS1 learning objectives.

Organization of Material

The GitHub repository is organized into the following directories and files, each containing resources that support the implementation, evaluation, and replication of the PDC activities.

```
Chapter8-HPU/
  COURSE_CALENDAR.xlsx
  Detailed_pre_and_Post_Syllabus/
  Evaluation_Data_and_Analysis/
  Plugged-In_Activities/
  Unplugged_Activities/
  README.md
```

COURSE_CALENDAR.xlsx

Purpose: This file contains the Spring 2026 course calendar for CSCI 2911/2916, outlining the weekly course schedule and instructional structure at Hawai'i Pacific University.

Detailed_Pre_and_Post_Syllabus

Purpose: This directory contains the course syllabus used before and after integrating the PDC activities and documents the changes introduced by the intervention.

Contents: Post-intervention course syllabus, including the course description, learning outcomes, prerequisites, grading policy, and schedule. The syllabus remains unchanged except for the integration of PDC activities. The schedule presents both pre- and post-intervention timelines, highlights intervention weeks, and includes a narrative summary of the changes.

Evaluation_Data_and_Analysis

Purpose: This directory contains the evaluation data and analysis materials used to assess the impact of the Parallel and Distributed Computing (PDC) activities.

Contents: Assessment data, anonymized survey responses, evaluation metrics, statistical analyses, and supporting documentation related to student learning outcomes and course effectiveness.

Plugged-In_Activities

Purpose: This directory contains the plugged-in programming activities used to introduce and reinforce PDC concepts.

Contents: Program source code, lab activity, supporting documentation, and instructional resources for the plugged-in (Parallel Sort Penny) Activity.

Unplugged_Activities

Purpose: This directory contains the unplugged activities (Flag Maker, Penny, MP) used to introduce foundational PDC concepts without requiring programming.

Contents: Activity guides, worksheets, instructional materials, facilitator resources, and supporting documentation for the unplugged PDC activities.

README.md

Purpose: This file provides an overview of the contents and organization of the directory.

Survey and Other Anonymized Data and Analysis

The Evaluation_Data_and_Analysis directory in the GitHub repository contains the de-identified and anonymized datasets, along with the analytical materials used to evaluate the effectiveness of the PDC interventions described in this chapter.

Chapter 9

CS1 Course Package – Montclair State University

***Focused Java Minimal-Infusion Model Centered on Flag Maker and Data Parallelism*†**

Jiayin Wang[1] and Michelle Zhu [2]

[1]Montclair State University, wangji@montclair.edu
[2]Kennesaw State University, mzhu10@kennesaw.edu

†**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

Abstract . . . 269
9.1 Introduction . . . 272
9.2 How CS1 was changed and PDC topics integrated . . . 272
9.2.1 Integrated Syllabi (Pre and Post) . . . 272
9.2.2 Highlights . . . 273
9.2.3 Challenges . . . 274
9.3 New Unplugged Activities . . . 275
9.3.1 Flag Maker . . . 275
9.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 275
9.3.1.2 Implementation Details . . . 275
9.3.1.3 Experience and Teaching Tips . . . 276
9.3.1.4 Data and Analysis . . . 276
9.4 New Plugged-in Activities . . . 278
9.5 Course-wide Pre and Post Course Surveys . . . 278
9.6 Other Activities and Ideas . . . 282
9.7 Conclusion . . . 282
Appendix . . . 284

Chapter tables

9.1 Relevant PDC topics and Bloom's levels . . . 270
9.2 MSU CS1 gender distribution . . . 271
9.3 MSU CS1 race/ethnicity distribution . . . 271
9.4 MSU CS1 current class standing distribution . . . 271
9.5 Summary comparison of CS1 before and after PDC integration. . . . 273
9.6 Post-integration CS1 course schedule with PDC integration highlighted. . . . 274
9.7 Summary of Flag Maker post-activity survey responses ($n = 8$). . . . 277
9.8 Themes from open-ended Flag Maker activity responses. . . . 278
9.9 CS1 survey results for basic programming skills. . . . 280
9.10 CS1 survey results for PDC concepts. . . . 280
9.11 CS1 survey results for attitudes and beliefs. . . . 280
9.12 Matched pre- and post-course survey results for basic computing concepts in CS1. . . . 281
9.13 Matched pre- and post-course survey results for PDC concepts in CS1. . . . 281
9.14 Matched pre- and post-course survey results for attitudes and perceptions in CS1. . . . 282

Abstract

This course exemplar describes the integration of an unplugged data-parallelism activity in CSIT 111, *Fundamentals of Java Programming*, a CS1-level Java programming course at Montclair State University. The activity uses a paper-based Flag Maker task, supported by short browser-based animations, to introduce students to foundational parallel and distributed computing (PDC) concepts without requiring specialized hardware, parallel programming libraries, or a separate programming assignment. Students physically model processors that color a grid representation of the flag of Mauritius under different task-partitioning strategies. By comparing a single-processor execution, a two-processor execution, a four-processor stripe-based execution, and a four-processor column-based execution, students experience sequential execution, parallel execution, speedup, synchronization, load imbalance, race conditions, and pipelining in a concrete and accessible way. The activity is embedded within the existing CSIT 111 course structure and is accompanied by a pre-activity quiz, post-activity quiz, a pre-course survey at the beginning of the semester, and a post-course survey at the end of the semester. Paired pre- and post-survey responses were consistent with gains in students' self-reported familiarity with parallel processing and distributed computing and their understanding of PDC concepts, although the small sample size limits the strength of these conclusions.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Revel for Java Software Solutions, Foundations of Program Design, 10th Edition by Lewis & Loftus[37]

Relevant core courses: CS1: CSIT 111 Fundamentals of Java Programming

Pre-requisites: CS0: CSIT 104 Python Programming I

Relevant PDC topics: Table 9.1 lists the integrated PDC topics with bloom's classification (Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR)). Because CS1 uses Flag Maker activity as an unplugged conceptual exercise, the Bloom levels are limited to Remember (RE) and Understand (UN).

Student Learning outcomes (SLOs):

1. Students will be able to distinguish sequential execution from parallel execution.
2. Students will be able to explain how dividing work among multiple processors can affect completion time, speedup, scalability, and load balance.
3. Students will be able to recognize when an algorithmic solution can benefit from the use of multiple threads that communicate.
4. Students will be able to identify common PDC issues illustrated by the activity, including synchronization, race conditions, contention, dependencies, idle time, and pipelining.

Table 9.1: Relevant PDC topics and Bloom's levels

PDC Concept	Chapter Section 9.3.1
Data parallelism	UN
Sequential vs. parallel execution	UN
Speedup	RE
Scalability	RE
Load balancing and idle time	UN
Synchronization	RE
Contention for shared resources	RE
Race conditions	RE
Pipelining	RE
Distributed task partitioning	UN

Context for use: Montclair State University, located 12 miles west of midtown Manhattan, is New Jersey's largest Hispanic-Serving Institution (HSI) and a public R2 doctoral research university enrolling approximately 23,000 students. The Flag Maker activity was implemented in CSIT 111, *Fundamentals of Java Programming*. CSIT 111 is a CS1-level introductory Java programming course offered by the School of Computing and required for both the BS in Computer Science and the BS in Information Technology. Table 9.2, Table 9.3, and Table 9.4 summarize the demographic characteristics of students who completed the pre-course survey in Spring 2025 and Fall 2025. Specifically, the tables report the distribution of students by gender, race/ethnicity, and current class standing, providing context for interpreting the survey results across the two semesters.

The course is taught over a 14-week semester, with approximately seven sections offered each semester and an average class size of 32 students. The course introduces fundamental programming concepts, including primitive data types, expressions, conditionals, loops, arrays, classes, objects, and object-oriented program design. The revised version of CSIT 111 retains the standard CS1 structure while adding explicit exposure to data parallelism. The updated course overview states that students are introduced to the fundamentals of data parallelism and to how programs can efficiently use modern multi-core processors. The revised learning outcomes also include recognizing when an algorithmic solution can benefit from the use of multiple communicating threads. The Flag Maker activity was placed in Class 23 out of 24 classes as a data parallelism activity. By this point in the course, students had already encountered core CS1 ideas. In this implementation, the activity is used as an unplugged in-class exercise supported by browser-based animations. Students use printed grids, coloring materials, and timing data to model processors working on the same task under different partitioning strategies. The activity is designed for instructors who want a low-barrier way to introduce PDC concepts in CS1 without displacing major Java programming content.

Table 9.2: MSU CS1 gender distribution

Gender	SP25 ($n = 13$)	FA25 ($n = 34$)
Female	3	8
Male	10	25
Not reported	0	1
Total	13	34

Table 9.3: MSU CS1 race/ethnicity distribution

Race/Ethnicity	SP25 ($n = 13$)	FA25 ($n = 34$)
Asian / Pacific Islander	0	1
Black/African American	4	12
Hispanic	4	9
Middle Eastern or North African	0	3
White	3	5
Declined to specify	1	3
Not reported	1	1
Total	13	34

Table 9.4: MSU CS1 current class standing distribution

Current Class Standing	SP25 ($n = 13$)	FA25 ($n = 34$)
First year (freshman)	7	12
Sophomore	2	13
Junior	2	5
Senior	2	4
Total	13	34

9.1 Introduction

Parallel and distributed computing (PDC) is now central to modern computing, yet many CS1 courses still introduce programming primarily through sequential execution. Students learn how to write programs that proceed step by step, but they often have limited opportunities to reason about how work can be divided, executed concurrently, and coordinated across multiple processing units. Introducing these ideas early can help students develop a more accurate mental model of contemporary computing systems.

At the same time, CS1 courses are already full of essential topics, including variables, expressions, conditionals, loops, arrays, classes, objects, and problem-solving. Therefore, adding PDC content should not require removing core programming material or introducing advanced parallel programming tools too early. The goal of this activity is to provide a lightweight, accessible entry point into PDC through an unplugged classroom exercise that complements, rather than competes with, the existing Java curriculum.

The Flag Maker activity introduces data parallelism by asking students to physically model processors coloring a grid representation of the flag of Mauritius. By comparing single-processor and multiprocessor versions of the same task, students observe how parallel execution can reduce completion time while also creating new challenges, such as synchronization, contention, race conditions, and coordination overhead. Browser-based animations are used to reinforce the connection between the physical activity and the underlying computing concepts.

This section presents the design, implementation, and assessment of the Flag Maker activity in CS1. The activity is assessed through pre- and post-activity quizzes, while broader course-level changes are examined through beginning-of-semester and end-of-course surveys.

9.2 How CS1 was changed and PDC topics integrated

The integration strategy preserved the original CS1 structure while adding a focused PDC learning experience. The pre-integration syllabus presented CS1 as a Java programming course covering digital computers, Java syntax and semantics, algorithms, logic, design, testing, documentation, and ethics in computing. Its course outcomes focused on program creation, assignments and expressions, Java library use, classes and objects, structured programming, arrays, objects, conditionals, and loops. The revised syllabus retained these core CS1 goals but added explicit PDC language in the course overview and learning outcomes. In particular, the revised course overview states that the course introduces the fundamentals of data parallelism and prepares students to understand how programs can efficiently use modern multi-core processors. The revised student learning outcomes also add an outcome requiring students to recognize when an algorithmic solution can benefit from multiple threads that communicate.

9.2.1 Integrated Syllabi (Pre and Post)

The pre- and post-integration syllabi for CS1 show that the course remained a CS1 Java programming course. The main course description and programming goals were not

substantially changed. Both versions emphasize Java syntax and semantics, programming logic, algorithmic problem solving, conditionals, loops, arrays, classes, objects, and object-oriented program design. The purpose of the integration was therefore not to redesign the entire course, but to introduce PDC concepts within the existing CS1 structure.

The main differences between the pre- and post-integration syllabi were structural. The earlier version of CS1 was organized as a 16-week course with 75-minute class meetings, while the revised version was organized as a 14-week course with 85-minute class meetings. In addition, the in-class lab assignments were combined and reduced from 10 labs to 6 labs. This revised structure created space for a lightweight PDC intervention without replacing the core Java programming sequence. The key new integration point is Module 9, *Parallel Computing*, where the Flag Maker data parallelism activity was implemented as an in-class unplugged activity supported by browser-based animations and pre- and post-activity assessments.

Table 9.5: Summary comparison of CS1 before and after PDC integration.

Syllabus Element	**Pre-Integration Version**	**Post-Integration Version**
Primary language	Java	Java
Semester format	16 weeks, 75 minutes per class	14 weeks, 85 minutes per class
Core CS1 content	Java syntax, algorithms, logic, conditionals, loops, arrays, classes, objects, and object-oriented design	Java syntax, algorithms, logic, conditionals, loops, arrays, classes, objects, object-oriented design, **inheritance**, **PDC**
Lab structure	10 in-class lab assignments	6 combined in-class lab assignments
PDC content	Not explicitly included as a course module	**Module 9: Parallel Computing**
PDC activity	Not applicable	Flag Maker data parallelism activity with animations
PDC assessment	Not applicable	Pre-activity quiz, post-activity quiz, and course-level pre- and post-surveys

Table 9.6 summarizes the post-integration course schedule. The PDC component is intentionally limited in scope and placed near the end of the semester, after students have encountered the core programming constructs needed to reason about sequential execution, loops, conditionals, arrays, objects, and problem decomposition. The highlighted row shows the main syllabus-level integration of PDC content.

Overall, the syllabus integration was deliberately modest. The post-integration version preserved the traditional CS1 programming sequence while adding a short PDC module and an unplugged Flag Maker activity. This design allowed students to encounter data parallelism and related PDC concepts without requiring a separate parallel programming unit or reducing coverage of core Java topics.

9.2.2 Highlights

The CS1 integration is intentionally lightweight. It does not require parallel hardware, specialized software, or a new programming environment. Students use paper grids and coloring materials to physically model how a computer might execute a task under different processor configurations. The instructor then uses short browser-based animations to reinforce the mapping between the physical activity and the computing concepts.

Table 9.6: Post-integration CS1 course schedule with PDC integration highlighted.

Module	Week	Activities	Assignments
Module 1: Introduction	Week 1	Slides: Class 1–2	Homework 1
Module 2: Data and Expression	Weeks 2–3	Slides: Class 3–4	Lab 1; Homework 2
Module 3: Using and Writing Classes and Objects	Weeks 3–5	Slides: Class 6–8	Lab 2; Homework 3 and 4
Module 4: Conditionals and Loops	Weeks 6–7	Slides: Class 10–12, 14	Lab 3; Homework 5; Lab 4; Homework 6
Module 5: Midterm Exam	Week 8	Slides: Class 16; Midterm Review; Midterm Exam	None
Module 6: Object-Oriented Design	Week 9	Slides: Class 17–18	Lab 5; Homework 7
Module 7: Arrays	Week 10	Slides: Class 20–21	Lab 6; Homework 8
Module 8: Inheritance	Weeks 11–12	Slides: Class 22	Homework 9
Module 9: Parallel Computing	**Week 12**	**Slides: Class 23; Flag Maker data parallelism activity; animations**	**Pre-activity quiz and post-activity quiz**
Module 10: Final Exam	Weeks 13–14	Slides: Class 24; Final Review; Final Exam Part 1 and 2	None

The activity was designed to fit within a normal class period. Students first complete a pre-activity quiz. The instructor then introduces the flag-coloring rules and facilitates several timed scenarios. After each scenario, students discuss what happened, why performance changed, and what new problems appeared as more processors were introduced. The activity concludes with animation-based reinforcement and a post-activity quiz.

9.2.3 Challenges

The main challenge was classroom logistics. The instructor had to prepare enough printed grids, markers or crayons, timing devices, and grouping instructions. Because each scenario used a different number of students per group, the instructor needed to decide group sizes before class and prepare extra copies for uneven enrollment. It was also important to explain that the pre-activity quiz was diagnostic rather than punitive, since students might not yet know the PDC terms being assessed. Finally, the instructor needed to reserve time for discussion after each scenario. Without structured reflection, students might have experienced the activity as only a coloring exercise rather than as a model of parallel computation.

Another challenge was the lengthy IRB approval process. Project activities were delayed while approval was pending, and because the approved protocol did not permit us to offer bonus points as an incentive, student participation in the research activities was relatively low. Administrative support was also needed to propagate the intervention consistently across all sections. The course was taught by a mix of full-time faculty and part-time lecturers, which made securing buy-in from every instructor and maintaining consistent content delivery an ongoing challenge.

9.3 New Unplugged Activities

9.3.1 Flag Maker

9.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

In CS1, the Flag Maker activity was adopted as an unplugged in-class data parallelism exercise supported by browser-based animations. Students acted as processors and colored a grid representation of the flag of Mauritius under different task-partitioning strategies, including single-processor, two-processor, four-processor stripe-based, and four-processor column-based execution.

The activity required no prior knowledge of parallel programming. Students only needed basic CS1-level understanding of sequential execution, loops, conditionals, and task decomposition. This made the activity appropriate for an introductory programming course because students could reason about parallel execution before learning formal multithreading APIs.

Based on the classroom implementation, the main learning goal was conceptual understanding rather than programming implementation. Students were expected to distinguish sequential and parallel execution, explain how data can be partitioned across processors, compare completion time across different numbers of processors, and recognize common PDC issues such as load imbalance, contention, and pipelining. The activity also helped students connect familiar CS1 ideas, such as loops and row/column-based data organization, to the way modern computing systems divide and coordinate work.

For more information about the Flag Maker activity, please refer to Chapter 1, Section 1.3.1.

9.3.1.2 Implementation Details

The Flag Maker activity was implemented as a guided in-class unplugged exercise. The instructor used lecture slides to lead students through the activity sequence and to introduce the related PDC concepts after each scenario. The activity began with a pre-activity quiz, which was administered before any PDC concepts were introduced. After the activity, students completed a post-activity quiz with the same questions, followed by a short feedback survey about the Flag Maker activity.

During the activity, students were organized into groups of different sizes depending on the scenario. For the single-processor scenario, each group included one student acting as the processor and one student acting as the timer. For the two-processor and four-processor scenarios, additional students were assigned as processors according to the task-partitioning strategy. Each group received printed flag grids and crayons. The student responsible for timing used their own phone as a stopwatch.

Each group was assigned a group number. After each scenario, the instructor recorded the completion time for each group on the whiteboard. This allowed the class to compare the time costs across groups and across different processor configurations. The timing results were mainly used to support discussion of sequential execution, parallel execution, speedup, scalability, and load balancing.

The instructor then used the follow-up questions and explanation slides to connect each

scenario to additional PDC concepts. Scenario 1 supported discussion of sequential processing and single-processor bottlenecks. Scenario 2 introduced task division and possible race conditions when processors are assigned overlapping work. Scenario 3 supported discussion of speedup, scalability, synchronization, and idle time caused by load imbalance. Scenario 4 introduced an alternative column-based partitioning strategy and was used to discuss boundary management, contention for shared resources, dependencies, distributed task partitioning, and pipelining.

The activity concluded with browser-based animations that reinforced the concepts demonstrated in the unplugged exercise. The animations helped connect the physical coloring task to computational models of parallel execution and provided a visual bridge between the classroom activity and PDC terminology.

9.3.1.3 Experience and Teaching Tips

- Print the flag grid at less than half of a letter-size page to reduce coloring time. This helps preserve more class time for comparing timing results and discussing the PDC concepts.
- Tell students before the pre-quiz that the questions are intended to measure prior knowledge and that incorrect answers are expected.
- Keep the coloring rules strict. If students color multiple squares at once, the model no longer represents processor-level work accurately.
- Use visible timing data. Write each scenario's completion time on the board so students can compare observed speedup with expected speedup.
- Ask students why the four-processor scenario may not produce exactly a fourfold improvement. This opens discussion of human variation and coordination overhead.
- When discussing contention, use classroom examples such as two processors needing the same marker or trying to work on the same grid boundary.
- Use Scenario 4 to show that partitioning strategy matters. The same number of processors can behave differently depending on how the data are divided.
- Use the animations after students complete the unplugged activity, not before. This preserves the discovery experience and then provides formal reinforcement.

9.3.1.4 Data and Analysis

A post-activity survey was administered after the Flag Maker activity to collect students' perceptions of the activity, group learning, conceptual understanding, engagement, and instructional support. The survey contained Likert-scale items and open-ended questions. A total of eight valid student responses were included in this analysis. Because of the small sample size, the results are interpreted descriptively rather than as inferential evidence.

For the Likert-scale items, responses were coded as follows for summary purposes: Neutral = 4, Somewhat agree = 5, Agree = 6, and Strongly agree = 7. No student selected a disagree option for any of the Likert-scale items. Positive responses were defined as Somewhat agree, Agree, or Strongly agree. Table 9.7 summarizes the results by survey dimension.

The strongest results were related to instructional support. Students rated the instructor's preparation and effort especially highly, with both items receiving a mean of 5.88 out

Table 9.7: Summary of Flag Maker post-activity survey responses ($n = 8$).

Survey Dimension	Items Included	Mean	Positive Responses
Peer and group learning	Explaining material, receiving explanations, group discussion, group contributions	5.19	96.9%
Engagement and preference	Having fun, preferring a class with the activity, focus, effort	5.19	87.5%
PDC and course learning	Confidence, understanding of parallel computing, understanding of loops, connection to programming assignment	4.91	81.3%
Student contribution	Valuable contribution to the group	5.25	87.5%
Instructional support	Instructor preparation, effort, enthusiasm, and availability of instructor or TA	5.72	93.8%

of 7. The instructor or TA availability item also received a high mean score of 5.63. These results suggest that students perceived the activity as well organized and supported during implementation.

The group-learning results were also positive. All students agreed at least somewhat that having the material explained by group members improved their understanding, and all students agreed at least somewhat that group discussion contributed to their understanding of parallel computing. This supports the design choice of using an unplugged collaborative activity rather than presenting PDC concepts only through lecture.

Students also reported that the activity supported their understanding of PDC concepts. The item "The activity increased my understanding of parallel computing" had a mean of 5.13, with 87.5% positive responses. The item about understanding loops had a lower mean of 4.50, suggesting that students viewed the activity more strongly as a parallel-computing activity than as a direct reinforcement of loop concepts. This is reasonable because the CS1 version used the Flag Maker primarily as an unplugged PDC activity rather than as a plugged-in Java programming assignment.

Open-ended responses provide additional evidence that students connected the activity to PDC ideas. Several students described learning that multiple processors, represented by multiple people working together, can complete a task faster than a single processor. One student specifically mentioned pipelining and CPU performance, while another noted that changing how a system works can greatly increase efficiency. These comments suggest that the activity helped some students move beyond the surface-level coloring task and reason about system-level performance.

Overall, the Flag Maker post-activity survey suggests that students found the activity engaging, collaborative, and useful for understanding parallel computing. The results are exploratory because of the small number of responses, but they provide encouraging evidence that a short unplugged activity can help CS1 students reason about parallel execution, speedup, pipelining, and the role of coordination in multiprocessor systems.

Table 9.8: Themes from open-ended Flag Maker activity responses.

Theme	Representative Evidence	Interpretation
Parallel execution and speedup	Students noted that multiple people working together completed the coloring task faster than one person working alone.	The activity made the benefit of parallel execution concrete.
Pipelining and system performance	One response identified pipelining as an interesting concept related to improving CPU performance.	Some students connected the activity to broader computer architecture ideas.
Efficiency through system design	One response noted that changing how a system works can greatly increase efficiency.	Students recognized that task organization affects performance.
Positive activity design	Several students wrote that no major improvement was needed or that the activity was well planned.	Students generally perceived the activity as clear and effective.

9.4 New Plugged-in Activities

No new plugged-in PDC programming activity was added to CS1 because the Java programming sequence at our institution is divided into two sequential courses: CSIT 111, *Fundamentals of Programming I*, and CSIT 112, *Fundamentals of Programming II*. CSIT 111 focuses on introductory programming foundations, including variables, expressions, conditionals, loops, arrays, classes, objects, and basic problem solving. At this stage, students are still developing basic programming fluency, so requiring them to implement PDC concepts with threads or other parallel programming tools would likely create too much cognitive load. Therefore, the Flag Maker activity was designed as an unplugged conceptual activity that introduces core PDC ideas without requiring parallel programming. Programming-based PDC activities are instead placed in our CS2, where students have stronger Java preparation and can work with plugged-in activities such as multithreaded image processing.

This CS1 activity also provides a conceptual foundation for the CS2 PDC activities described in Chapter 18. In Chapter 9, students are introduced to data parallelism through the unplugged Flag Maker activity, where they reason about task partitioning, parallel execution, speedup, load balancing, and coordination without the additional complexity of concurrent programming. In Chapter 18, these ideas are extended into a programming context through Greenfoot event handling and multithreaded image processing. Together, the two course packages provide a progression from conceptual exposure to data parallelism in CS1 to implementation and experimentation with PDC concepts in CS2.

9.5 Course-wide Pre and Post Course Surveys

To examine changes associated with integrating parallel and distributed computing (PDC) concepts throughout the CS1 course, we administered matched pre- and post-course surveys

at the beginning and end of the semester.

The survey measured students' self-reported understanding of fundamental programming concepts, PDC concepts, and attitudes toward computing. Students rated their level of understanding of the following programming concepts:

1. Data/Variable Types
2. Arithmetic Expressions and Operators
3. Branching
4. Loops
5. Calling Functions/Methods
6. Arrays
7. Writing Static Functions
8. Writing Classes
9. Writing Methods
10. Creating Objects

To evaluate students' understanding of PDC, they also rated their familiarity with the following concepts:

1. Parallel Processing
2. Distributed Computing
3. Event Handling

Finally, students responded to the following attitude statements:

A1 In general, my interest in computing is high.
A2 My interest in learning about parallel and distributed computing is high.
A3 In general, I believe learning about parallel and distributed computing is important.
A4 I believe I have a good understanding of computing.
A5 I believe I have a good understanding of parallel and distributed computing.

All survey items were measured using a seven-point Likert scale. Programming and PDC concept questions ranged from *1 = No Experience at All* to *7 = Extremely Experienced*, while attitude questions ranged from *1 = Strongly Disagree* to *7 = Strongly Agree*.

As shown in Table 9.9, the comparison between the no-intervention semester and the Flag Maker intervention semester suggests that the Flag Maker activity did not show evidence of reduced self-reported progress in basic programming skills. Students in both semesters reported gains from pre-survey to post-survey across all basic programming categories. The no-intervention semester showed slightly larger gains for several introductory topics, such as data types, arithmetic expressions, branching, and writing static functions. However, the intervention semester had comparable post-survey means, and in some cases slightly higher post-survey means, such as loops, writing methods, and creating objects. This suggests that adding the unplugged Flag Maker activity did not reduce students' perceived progress in the core CS1 programming content.

Table 9.9: CS1 survey results for basic programming skills.

	No intervention			Intervention		
Basic Programming Skill	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Data types	3.12	4.50	1.38	3.33	4.33	1.00
Arithmetic expressions and operators	2.75	4.62	1.88	3.00	4.44	1.44
Branching	2.75	4.50	1.75	3.00	4.44	1.44
Loops	3.00	4.75	1.75	3.22	4.78	1.56
Calling functions/methods	3.38	3.88	0.50	3.56	4.00	0.44
Arrays	2.50	3.62	1.12	2.78	3.62	0.85
Writing static functions	2.50	4.12	1.62	2.78	4.11	1.33
Writing classes	3.50	4.50	1.00	3.67	4.44	0.78
Writing methods	2.88	4.38	1.50	3.11	4.44	1.33
Creating objects	3.12	4.88	1.75	3.33	4.89	1.56

For PDC concepts, both semesters showed gains from pre-survey to post-survey, as shown in Table 9.10. The intervention semester had higher post-survey means than the no-intervention semester for all three PDC-related concepts: parallel processing, distributed computing, and event handling. Although the mean gains were larger in the no-intervention semester, students in the intervention semester entered with higher pre-survey ratings. The higher post-survey means in the Flag Maker semester are consistent with slightly stronger self-reported understanding of these PDC-related topics at the end of the course.

Table 9.10: CS1 survey results for PDC concepts.

	No intervention			Intervention		
PDC Concept	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Parallel processing	1.29	2.62	1.34	1.75	2.78	1.03
Distributed computing	1.14	2.75	1.61	1.62	3.00	1.38
Event handling	1.57	3.38	1.80	2.00	3.44	1.44

The attitude and belief results were mixed, as summarized in Table 9.11. Students in both semesters reported high interest in computing and high belief in the importance of learning PDC. Because these ratings were already relatively high at the beginning of the semester, there was limited room for large improvement. The intervention semester showed similar or slightly lower gains in general interest and perceived importance, but students' self-reported understanding of PDC was slightly higher at the end of the intervention semester than in the no-intervention semester. Overall, the comparison is consistent with increased PDC awareness during the intervention semester while showing no indication of reduced progress in basic programming skills.

Table 9.11: CS1 survey results for attitudes and beliefs.

	No intervention			Intervention		
Attitude / Belief	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Interest in computing is high	5.00	5.88	0.88	5.00	5.67	0.67
Interest in learning about PDC is high	4.86	5.00	0.14	4.88	4.89	0.01
Learning about PDC is important	4.14	5.38	1.23	4.38	5.33	0.96
Good understanding of computing	5.14	4.62	-0.52	5.25	4.67	-0.58
Good understanding of PDC	2.14	3.38	1.23	2.50	3.56	1.06

In addition to the comparison between the no-intervention and intervention semesters, we conducted a more detailed analysis of the matched pre- and post-course survey responses from the intervention semester. Matched responses were analyzed using the nonparametric Wilcoxon signed-rank test. Statistical significance was evaluated at $\alpha = 0.05$, and effect sizes were computed using Cliff's delta (δ). Tables 9.12–9.14 report the pre- and post-course means, mean differences, Wilcoxon statistics (W), p-values, statistical significance, Cliff's delta (δ), and effect size interpretations.

Table 9.12 shows that students reported improvement across all basic computing concepts during the intervention semester. The largest mean gains were observed for loops and creating objects, followed by arithmetic expressions and operators, branching, writing static functions, and writing methods. Branching and loops reached statistical significance at the $\alpha = 0.05$ level, suggesting that students made measurable progress in these foundational CS1 topics. Several other items did not reach statistical significance; however, their positive mean differences and medium to large effect sizes were consistent with improvement in students' perceived programming knowledge.

Table 9.12: Matched pre- and post-course survey results for basic computing concepts in CS1.

Item	**Pre**	**Post**	**Diff**	W	p**-value**	**Sig.**	**Cliff's** δ	**Interp.**
Data/Variable Types	3.33	4.33	1.00	3.5	0.188	No	-0.28	small
Arithmetic Expressions and Operators	3.00	4.44	1.44	4.5	0.062	No	-0.49	large
Branching	3.00	4.44	1.44	2.5	0.039	Yes	-0.43	medium
Loops	3.22	4.78	1.56	0.0	0.016	Yes	-0.49	large
Calling Functions/Methods	3.56	4.00	0.44	1.0	0.500	No	-0.16	small
Arrays	2.50	3.62	1.12	7.5	0.188	No	-0.39	medium
Writing Static Functions	2.78	4.11	1.33	4.0	0.055	No	-0.40	medium
Writing Classes	3.67	4.44	0.78	1.5	0.188	No	-0.25	small
Writing Methods	3.11	4.44	1.33	1.5	0.094	No	-0.46	medium
Creating Objects	3.33	4.89	1.56	0.0	0.062	No	-0.44	medium

Table 9.13 summarizes the matched pre- and post-course survey results for PDC-related concepts. Students reported higher post-course understanding for all three PDC concepts: parallel processing, distributed computing, and event handling. Although none of these gains reached statistical significance, the effect sizes were medium for parallel processing and event handling and large for distributed computing. This pattern is consistent with increased student awareness of PDC concepts during the intervention semester, although the observed changes were not statistically significant and should be interpreted cautiously given the small sample size.

Table 9.13: Matched pre- and post-course survey results for PDC concepts in CS1.

Item	**Pre**	**Post**	**Diff**	W	p**-value**	**Sig.**	**Cliff's** δ	**Interp.**
Parallel Processing	1.75	2.62	0.88	2.0	0.250	No	-0.42	medium
Distributed Computing	1.62	2.88	1.25	0.0	0.125	No	-0.50	large
Event Handling	2.00	3.12	1.12	7.5	0.344	No	-0.45	medium

Table 9.14 presents the results for students' attitudes and perceptions. Students entered the course with relatively high interest in computing and high interest in learning about PDC, leaving limited room for additional growth in these areas. Interest in computing increased slightly, while interest in learning PDC remained unchanged. Students' perceived importance of learning PDC and their self-reported understanding of PDC were higher at post-survey, with PDC understanding increasing from 2.50 to 3.50 and showing a medium effect size; however, these changes were not statistically significant.

Table 9.14: Matched pre- and post-course survey results for attitudes and perceptions in CS1.

Item	Pre	Post	Diff	W	p-value	Sig.	Cliff's δ	Interp.
Interest in Computing	5.00	5.50	0.50	5.0	0.344	No	-0.25	small
Interest in Learning PDC	4.88	4.88	0.00	7.5	1.000	No	0.06	negligible
Importance of Learning PDC	4.38	5.12	0.75	2.0	0.250	No	-0.25	small
Understanding of Computing	5.25	4.75	-0.50	1.5	0.375	No	0.31	small
Understanding of PDC	2.50	3.50	1.00	2.5	0.078	No	-0.41	medium

Note. Responses were coded on a seven-point Likert scale. Diff represents the post-course mean minus the pre-course mean. Wilcoxon signed-rank tests were two-sided, with statistical significance evaluated at $\alpha = 0.05$. Cliff's δ was calculated as $\delta(\text{pre}, \text{post})$; therefore, negative values indicate that post-course scores tended to be higher than pre-course scores. The matched sample size was $n = 9$ for most computing-concept items and $n = 8$ for items containing one missing matched response.

Because only eight to nine matched responses were available for each survey item, the statistical analyses had limited power. Therefore, the results should be interpreted as exploratory. Although only a small number of items reached statistical significance, the positive mean differences and moderate to large effect sizes across several programming and PDC concepts are consistent with improvement in students' perceived understanding during the intervention semester.

9.6 Other Activities and Ideas

No additional PDC activities were added to CS1 because the course schedule was already very tight. CS1 is the first course in the Java programming sequence, and most class time is needed for foundational programming topics. Adding more PDC activities would have required removing or reducing essential CS1 content.

For this reason, the Flag Maker activity was designed as a short unplugged activity that could introduce core PDC ideas without requiring major changes to the course structure. More programming-based PDC activities are better suited for CS2, where students have stronger Java preparation and more experience with object-oriented programming.

9.7 Conclusion

The CS1 Flag Maker activity demonstrates that PDC concepts can be introduced in a CS1 course through a short, accessible, unplugged intervention. The revised course preserves

the standard Java programming sequence while adding explicit exposure to data parallelism and multi-core thinking. Students experience the limitations of sequential execution, the performance benefits of parallel task partitioning, and the challenges that arise when multiple processors must coordinate their work. Survey results were consistent with gains in students' self-reported PDC understanding, providing preliminary support for integrating PDC concepts early in the programming curriculum.

The delays and low participation stemming from the IRB process argue for submitting protocols well before the intervention semester and for building alternative, IRB-permissible incentives into the initial application, such as raffles, or making the activity itself a regular in-class exercise so that research participation requires only consent to use data already being generated, rather than extra effort from students. Finally, consistent adoption across sections taught by both full-time faculty and part-time lecturers is best achieved by lowering the cost of participation: a plug-and-play instructor packet with slides, survey links, lesson plan, and the assessment instruments; a brief virtual training session at the start of the semester; and endorsement and acknowledgment from the course coordinator and department administration during department meetings, which converts individual instructor buy-in into a shared course-level commitment.

With these refinements, the Flag Maker activity can scale beyond a single instructor's classroom to a durable, department-wide component of the CS1 curriculum, offering a low-cost and transferable model for other institutions seeking to bring PDC concepts into introductory courses.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter9-MSU

Detailed Pre and Post Syllabus

The Parallel and Distributed Computing (PDC) intervention was implemented in CSIT 111, Fundamentals of Programming I, the CS1 course at Montclair State University. CSIT 111 is a Java-based introductory programming course covering fundamental programming concepts such as data and expressions, using and writing classes and objects, conditionals and loops, object-oriented design, arrays, and inheritance. The Spring 2025 semester served as the baseline semester before the PDC intervention was introduced. The Fall 2025 semester incorporated the Flag Maker unplugged activity as the PDC intervention.

The pre-intervention syllabus followed the standard CS1 structure and focused on introductory programming and object-oriented programming topics without an explicit PDC activity. The post-intervention syllabus preserved the existing CS1 course structure while integrating the Flag Maker activity into the later part of the semester. The Flag Maker activity was implemented on December 3, 2025, after students had developed basic programming knowledge needed to discuss computation, task division, and program execution. Through this activity, students explored sequential execution, parallel execution, data parallelism, speedup, scalability, synchronization, load balancing, race conditions, shared resources, and task partitioning. The pre- and post-course experience surveys were collected during the baseline and Flag Maker semesters to compare student experience, attitudes, and PDC-related learning outcomes. In this way, the PDC intervention was added to an otherwise standard CS1 sequence without displacing existing core programming content.

Organization of Material

The `chapter9-MSU` directory is organized as follows:

```
chapter9-MSU/
  COURSE_CALENDAR.md
  Detailed_Pre_and_Post_Syllabus/
  Evaluation_Data_and_Analysis/
  Plugged_Activities/
  Unplugged_Activities/
  README.md
```

COURSE_CALENDAR.md

Purpose: This file (available here) provides the Fall 2025 course calendar for CSIT 111 at Montclair State University.

Contents: A week-by-week schedule of course modules, class meetings, labs, homework assignments, examinations, breaks, and the Flag Maker PDC activity.

Usage: Primary reference for understanding where the PDC intervention was placed within the overall CS1 course sequence.

Detailed_Pre_and_Post_Syllabus

Purpose: This directory (available here) contains the pre- and post-intervention syllabus materials.

Contents: The Spring 2025 pre-PDC syllabus and the Fall 2025 post-PDC syllabus for CSIT 111.

Usage: Reference materials for comparing the baseline CS1 course structure with the semester in which the Flag Maker PDC activity was implemented.

Evaluation_Data_and_Analysis

Purpose: This directory (available here) contains the anonymized and aggregate evaluation materials used to assess the intervention.

Contents: Aggregate analysis outputs comparing the Spring 2025 baseline semester and the Fall 2025 Flag Maker semester, including student programming experience, attitudes and beliefs, and PDC-related survey results.

Usage: Source for evaluation summaries, comparison tables, and analysis of student learning and experience before and after the PDC intervention.

Plugged_Activities

Purpose: This directory (available here) documents that no plugged activity was implemented as part of this CS1 course package.

Contents: A short README noting that the CS1 intervention used the Flag Maker unplugged activity rather than a plugged programming activity.

Usage: Clarifies the activity structure for readers and directs them to the unplugged Flag Maker materials.

Unplugged_Activities

Purpose: This directory (available here) contains the unplugged PDC activity used in the CS1 course.

Contents: The Flag Maker activity materials, including the activity README and the instructor slide deck `CS1_MSU_FlagMaker_slides.pptx`.

Usage: Classroom materials for implementing the Flag Maker activity and introducing PDC concepts through a guided, hands-on, non-programming exercise.

README.md

Purpose: Quick reference documentation for the MSU CS1 course package.

Contents: Overview of the course package, activity summary, and repository organization.

Usage: Entry point for users to understand the module's organization and purpose.

Survey and Other Anonymized Data and Analysis

On the GitHub link, the directory "Evaluation_Data_and_Analysis" contains anonymized aggregate analysis files for evaluating the PDC intervention. The Spring 2025 semester served as the baseline semester, and the Fall 2025 semester included the Flag Maker activity. The analysis materials include comparison outputs for student programming experience, attitudes and beliefs, and PDC-related learning outcomes. Raw student-level survey files and open-ended responses are not included in the public repository in order to protect student privacy.

PART II - Computer Science 2 (CS2)

Chapter 10

Common CS2 Parallel and Distributed Computing Activities and Lessons

***Shared Activity Families, Adoption Guidance, and Evidence*†**

David P. Bunde (Knox College),
April R. Crockett (Tennessee Technological University),
Srishti Srivastava (University of Southern Indiana),
Michelle Zhu (Kennesaw State University),
Jaime Spacco (Knox College),
Mary L. Smith (Hawai'i Pacific University),
Charlotte Gruner (Casper College),
Jiayin Wang (Montclair State University),
Chris Bourke (University of Nebraska–Lincoln),
Xiaoyuan Suo (Webster University),
Peter Maher (Webster University),
Gerald C. Gannod (Tennessee Technological University),
Alan Sussman (University of Maryland, College Park),
Neena Thota (University of Massachusetts Amherst),
Charles Weems (University of Massachusetts Amherst),
Ramachandran Vaidyanathan (Louisiana State University), and
Sushil K. Prasad (University of Texas at San Antonio)

†**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Abstract

This chapter distills some common CS2 activities, implementation strategies, challenges, and lessons learned across the eight institutional CS2 exemplars in this volume. It serves as a shared resource chapter between the project overview in Chapter 0 and the institution-specific CS2 chapters 11–18 that follow. Rather than presenting a local course implementation, the chapter identifies activity families employed across multiple institutions and reusable guidance that cut across TNTECH, Knox, USI, UNL, Webster, Casper, HPU, and MSU. Its purpose is to help instructors locate activities that can be adopted as a single class activity, a short module, a laboratory exercise, or part of a broader course-level infusion of parallel and distributed computing (PDC) into CS2, while ensuring that basic CS2 learning outcomes are preserved.

The activities summarized here introduce versions of core PDC ideas, including event driven programming, task parallelism, shared memory parallelism, and data parallelism. These ideas are connected to familiar CS2 topics such as recursion, nested loops, multi-dimensional arrays, and algorithmic complexity analysis. Recurring activity families include unplugged activities such as Overly Complicated Machine; and programming activities involving fork-join parallelism, sorting algorithm comparison and parallel image processing.

The chapter points readers to institutional chapters and their repositories, appendices, assessment instruments, student feedback, and classroom evidence so they can select and adapt materials that match their own teaching context. The repositories will be periodically curated and updated with instructional material and evidence.

Across institutions, a shared lesson is embedding PDC concepts within existing CS2 learning goals rather than treating them as disconnected advanced content.

10.1 Introduction

This chapter describes shared CS2 activities and lessons that emerged across the institutional exemplars. Unlike Chapters 11–18, which describe local course implementations, this chapter is organized around reusable activity families. Its purpose is to help instructors quickly identify activities that may fit their own CS2 course and then move to the institutional chapters for more detailed implementation features, adaptations, evidence, and repository resources.

The chapter is intended for selective, adoption-oriented reading. An instructor may use it to find a single unplugged activity, a short visualization, a programming assignment, an assessment instrument, or a starting point for broader CS2 infusion. The activities are deliberately connected to familiar CS2 topics.

Table 10.1 provides a quick activity finder for this chapter. The first column briefly describes each activity and points to the section where readers should begin. The next two columns highlight the PDC ideas and the CS2 connections they reinforce. The last column identifies institutional implementations and, where available, their estimates of instructional or student time.

Table 10.1: Activity finder for CS2 PDC activities described in this common chapter.

Activity and brief description	Primary PDC ideas	CS2 anchors	Duration / where used
Unplugged activity			
Overly Complicated Machine (OCM). Teams design a five-event Rube Goldberg–style chain in which each action triggers the next, then map the physical chain to events, responses, and event handlers. §10.3.1.	Event-driven programming; events, responses, and event handlers; inversion of control; reactive control flow; triggering and dependencies.	Control flow; methods/callbacks; classes/objects; GUI/event handling; state changes; framework-controlled execution.	Common activity: about 30–35 minutes. USI uses a fuller classroom sequence of about 46 minutes including introduction, reflection, and formalization; §13.3.1.
Plugged-in programming activities			
Fork–Join. Students move from simplified `fork/join` notation and dependency graphs to Java's Fork/Join framework, recursive tasks, reductions, and experiments with cutoff values. §10.4.2.	Task parallelism; recursive decomposition; fork/join; dependencies/precedence DAGs; work and span; overdecomposition; load balancing; task granularity; reductions; parallel overhead and speedup.	Recursion; divide-and-conquer; arrays; object-oriented programming and `RecursiveTask`; algorithm analysis; recursion trees; reductions.	Knox uses full frame work in approx. 100 min. of lecture plus labs; most labs fit in 70 minutes (§12.3). USI combines 40-min. unplugged (§13.3.2) with a week-long Fork/Join programming project (§13.4.2). Casper uses a 15 min. precedence-graph/ work–span activity(§17.3.3).
Sorting Algorithm Comparison. Students compare sequential and parallel sorting implementations, vary thread counts, measure execution time, and explain when parallelization helps or hurts. §10.4.3.	Shared-memory parallelism; independent work and dependencies; OpenMP; reductions; synchronization; overhead; speedup; scalability.	Sorting algorithms; recursion; divide-and-conquer; arrays/vectors; templates and comparators; Big-O analysis; timing and benchmarking.	TNTECH uses a 3-week C++ programming assignment (§11.3.1). UNL uses a Java parallel-sort lab (§14.4.2). Webster uses a C++ parallel Merge Sort. Casper uses a 2-hour parallel QuickSort (§17.4.4).

Continued on next page.

Table 10.1 continued.

Activity and brief description	Primary PDC ideas	CS2 anchors	Duration / where used
Parallel Image Processing. Students partition image-processing work across threads or processes, compare sequential and parallel kernels, verify correctness, measure runtime and speedup, and reason about synchronization and overhead. §10.4.4.	Data parallelism; task/data decomposition; concurrency versus parallelism; threads/process pools; race conditions; synchronization; task granularity; parallel overhead; speedup and scalability.	Nested loops and two-dimensional arrays; methods and classes; file and image I/O; image kernels/convolution; exception handling; thread-pool APIs; experimental performance analysis.	Common plan: about 135 minutes over two class sessions or a 115-minute workshop. USI uses a week-long team project with about 7.5 hours for image processing (§13.4.3); Webster uses a 50-minute module; and MSU uses two class sessions for lecture/demonstration and a box-blur lab (§18.3.2).

Additional CS2 PDC activities that are described in the institutional chapters, but not developed as full activities in this common chapter, are summarized in Table 10.2. These activities provide additional adoption options spanning unplugged exercises, event-driven programming, distributed systems, shared-memory programming, and performance experimentation.

Table 10.2: Additional institutional CS2 PDC activities not described as full activities in this common chapter.

Activity and brief description	Primary PDC ideas	CS2 anchors	Duration / where used
Unplugged, visual, or simulation activities			
Unplugged Penny Activity. Students search a penny collection sequentially for oldest penny, then repeat with multiple students and compare timings.	Sequential versus parallel processing; decomposition; synchronization; communication and coordination overhead; limits of speedup.	Searching; algorithm comparison; timing; collaborative problem solving.	Casper, 40-min. activity used in Week 1 §17.3.1.
Unplugged Coin Addition. Students simulate parallel array summation with a shared accumulator.	Race conditions; shared state; synchronization; reduction-style reasoning; coordination of parallel work.	Arrays; accumulation loops; variables; shared updates; summation.	Casper, 30-min. activity §17.3.2. Paired with the OpenMP Array Sum activity in §17.4.2.
Parallel Linked-List Access Animation. Interactive animation of concurrent operations occurring on a shared linked list.	Concurrent access; shared data; synchronization; race conditions.	Linked lists; shared data structures; operations on mutable structures.	Webster (Python), §16.3.
Distributed Dispatcher, Worker Simulation. Students role-play users, a central dispatcher, and worker processors. Users submit tasks to the dispatcher, who assigns them to workers that may be idle, running, waiting, or failed.	Task distribution; load balancing; communication overhead; synchronization; scalability; fault tolerance; distributed-system organization.	Task organization; state; scheduling; coordination among program components; system modeling.	Webster (Python), §16.3.
Plugged-in activities and extensions			
Greenfoot Event-Driven Game Project. Students design an interactive Java game whose behavior responds to events.	Event-driven programming; events and handlers; asynchronous and framework-driven control; callbacks; interaction among multiple event sources.	Classes and objects; inheritance; methods; GUI programming; APIs; program state; debugging; team programming.	USI 1-week team project §13.4.1. Casper 5-hours-per-student midterm project §17.4.3. MSU 2-week project including presentations §18.3.1.

Continued on next page.

Table 10.2 continued.

Activity and brief description	Primary PDC ideas	CS2 anchors	Duration / where used
Connection Pooling Lab. Java threads insert records into remote database; students implement/reuse pool of database connections maintained in a thread-safe blocking queue.	Distributed computing; multithreading; producer–consumer pattern; thread-safe shared-resource pooling; load generation; speedup.	SQL; JDBC; databases; queues; Java objects; database connectivity.	UNL, §14.4.1. Prior familiarity with SQL, JDBC, and queues is useful.
Basics of Parallel and Distributed Computing in Python. Short in-class Python exercises introduce PDC concepts.	Task decomposition; multiprocessing; concurrency; synchronization; race conditions; locks; deadlocks; parallel versus distributed execution.	Python programming; functions; data structures; process APIs; experimental programs.	Webster 50-min. Python class, §16.2.2 Module 1
Introduction to Performance Engineering. Students run/measure runtime of small experiment under different configurations, calculate/interpret speedup and efficiency, and explain observed performance trends and bottlenecks.	Performance measurement; benchmarking; speedup; efficiency; scalability; parallel overhead; compute- versus memory-bound behavior; bottleneck analysis.	Algorithm performance; empirical measurement; timing; experimental analysis; interpretation of program behavior.	Webster 50-min. Python class, §16.2.2 Module 3.
Zombie Outbreak Parallelization. Students modify a grid-based outbreak simulation, compare sequential and parallel execution, and discuss correctness and performance.	Task and domain decomposition; parallel execution; neighboring-region interaction; synchronization and communication; performance comparison.	Two-dimensional grids; loops; simulation; state updates; experimental timing.	Webster (Python), Chapter 16: optional/conditional homework requiring approximately 1–2 hours outside class. It is used in CS2 when it is not assigned in CS1.
OpenMP Array Creation and Array Sum Activities. Students parallelize array initialization, examine thread distribution, and measure speedup. They then parallelize summation, identify a race condition, and resolve it using partial results and synchronization.	Shared-memory parallelism; work sharing; threads; race conditions; reduction; synchronization; speedup and overhead.	C++ arrays; loops; dynamic memory; accumulators; compilation; `chrono` timing and benchmarking.	Casper, §17.4.1–17.4.2. Array Creation is designed as approximately a 2-hour lab. Array Sum can be distributed across several sessions or completed in a class period of at least 90 minutes.

The remainder of the chapter follows an adopter-facing structure. Section 10.2 summarizes how CS2 was changed across institutions and what PDC topics were integrated. Section 10.3.1 describes an unplugged activity. Sections 10.4.1, 10.4.2, 10.4.3, and 10.4.4 describe plugged-in activities. Section 10.5 summarizes the common course-wide pre/post survey structure. The conclusion distills cross-institution lessons for instructors who wish to adopt, adapt, or extend these materials.

10.2 How CS2 was changed and PDC topics integrated

We again see different institutions employ a variety of strategies in order to add the new activities/content. As with CS1, some institutions added new material to CS2 by exploiting slack that had existed in the course as previously taught. Others removed or de-emphasized some material that was either viewed as less relevant or was taught in a different course. Finally, some institutions were able to modify the coverage of existing topics in a way that covered new learning goals.

One difference we noticed between the changes to CS2 and the previously-discussed changes to CS1 was the greater emphasis on programming. For CS1, the new material was primarily conceptual, with heavy use of unplugged activities to teach PDC ideas without requiring students to write PDC code. The new CS2 courses still use unplugged activities, but they generally follow up with programming assignments.

Another difference between adoptions for CS1 and CS2 was that CS2 activities were generally less widely adopted. The Flagmaker Unplugged activity is the poster child for this observation, since it was so widely adopted, but even excluding that as an outlier, the CS2 activities were less widely adopted on average than the CS1 activities. This is perhaps not surprising since the material is more advanced and department curricula have had more time to diverge by the second course than the first. It has been shown previously that different institutions define "CS2" differently [48]. It does suggest that encouraging the modernization of CS2 (and higher-level courses) will be a greater challenge.

A third difference that we noticed is that some institutions made changes to their CS2 course using activities that were originally designed for CS1. In some cases, this was a temporary situation that reflected the instructor's course schedule and as a way to reach a cohort of students who had already taken CS1 before the project began. More interesting though, are cases where the module is expected to stay at the CS2 level. For example, Knox College created an assignment where students use the Greenfoot IDE as a framework for creating interactive code for their CS1 course §3.3.3 and both USI §13.4.1 and MSU §18.3.1 adopted the Greenfoot Event-Driven Game Project as part of their CS2 courses.

10.2.1 Challenges

Similar to CS1, one of the primary challenges in integrating PDC concepts into CS2 was finding space in an already crowded curriculum while ensuring that students continued to develop the foundational skills expected of the course. Unlike in CS1, where PDC concepts could often be introduced through short unplugged activities or conceptual demonstrations, changes to CS2 generally involved more substantial programming assignments that required additional instructional time. Instructors had to identify opportunities to incorporate PDC into existing topics such as recursion, sorting, algorithm analysis, and object-oriented programming without sacrificing the depth of coverage traditionally provided. The differences in CS2 curricula across institutions also made it difficult to develop activities that could be adopted without modification. Some institutions were able to make relatively minor changes to existing assignments, while others needed to introduce new modules or substantially adapt activities to fit their course structures.

A second challenge was determining the appropriate level of complexity for students who were still developing their programming skills. Although CS2 students have more programming experience than CS1 students, concepts such as task decomposition, concurrent execution, synchronization, and parallel performance introduce new levels of complexity. Students were often learning unfamiliar programming libraries and constructs, such as OpenMP, Java's fork–join framework, and Python multiprocessing, while simultaneously applying existing knowledge of recursion, algorithms, and data structures. We found that the activities needed to be simple enough for students to understand the underlying PDC concepts without becoming overwhelmed by implementation details, yet complex enough to demonstrate

meaningful applications of parallel computing. Simpler assignments sometimes lacked the depth needed to demonstrate the advantages of parallelism, while more realistic applications required additional preparation and programming experience. Instructors therefore had to balance accessibility with authenticity when selecting and sequencing activities.

Technical and logistical difficulties presented additional challenges. The programming-based nature of many CS2 activities required students to have compatible development environments, appropriate libraries, and access to necessary computing resources. Some assignments depended on external services or databases, creating potential problems involving access restrictions, API requirements, and network availability. Performance-based activities also required careful design because parallel execution does not necessarily produce faster results, particularly for small workloads where the overhead of creating and coordinating threads may outweigh the benefits. Instructors needed to ensure that students understood these outcomes rather than assuming that parallel implementations would always outperform sequential ones. Classroom logistics also varied by institution, with class size, instructional format, and available teaching support affecting which activities could be implemented effectively.

Finally, successful integration required additional instructor preparation, instructional support, and assessment development. Faculty and teaching assistants needed sufficient familiarity with the new concepts, programming tools, and assignments to support students effectively. In some cases, teaching assistants also needed a clearer understanding of the rationale for introducing PDC in CS2. Institutions frequently had to develop or adapt instructional materials because existing resources were not always suitable for students at this level. Evaluating student learning presented another challenge, particularly as generative AI tools made it easier for students to produce working code without necessarily understanding the underlying concepts. As a result, some assignments placed greater emphasis on experimentation, performance analysis, and explaining observed results. Collectively, these experiences suggest that integrating PDC into CS2 is feasible without a complete course redesign, but successful adoption requires careful attention to instructional sequencing, technical preparation, staff training, and assessment of conceptual understanding.

10.3 New Unplugged Activities

10.3.1 Overly Complicated Machine

Event handling is often a student's first encounter with a control flow that is not strictly sequential. Instead of statements executing in the order they are written, fragments of code lie dormant until some external occurrence such as a click, a keystroke, or a page finishing its load, causes them to activate. This unplugged activity builds an intuition for this inversion of control. Students are asked to design a whimsical, Rube Goldberg [42]-style contraption in which a chain of five "events" cascades toward a final goal, and then map that everyday cause-and-effect chain onto the vocabulary of event-driven programming using events, responses, and event handlers. A more detailed description of how this activity was implemented in a CS2 course at the University of Southern Indiana is given in Chapter 13, Section 13.3.1.

10.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

Problem description. This activity introduces the concept of event handling and event-based programming to students. Working in small teams, students brainstorm an "overly complicated machine," in which each step is triggered by the step before it. Each team's machine must contain exactly five distinguishable events and must accomplish some concrete final goal (for example, watering a plant, ringing a bell, or turning off a light). Teams sketch the machine on paper, explicitly labeling each of the five events, and then explain their design to a neighboring team. A guided reflection then draws the analogy between the physical chain reaction the students invented and the way a program reacts to events at run time. Every event in their machine has a corresponding response, just as every event in a graphical or web program must have an event handler that runs in response to it.

Prerequisites. The activity is intentionally lightweight in its technical prerequisites, which is what makes it a suitable introduction to the topic. Students should have completed an introductory programming sequence and be comfortable with basic control flow (sequencing, conditionals, and methods) so that the later contrast with non-sequential, event-driven execution is meaningful. No prior exposure to graphical user interfaces or any particular language's event model is required. Because the exercise is unplugged, students need no computers, development environment, or programming language during the activity itself.

Learning outcomes. After completing this activity, students will be able to:

- **Define an event** as an action or occurrence that signals an interaction with, or a change in the state of, a program.
- **Define an event handler** as the code that is written to react to a specific event and produce the appropriate response.
- **Distinguish an event from its response**, recognizing that events and handlers form pairs and that a program must supply a handler for each event it cares about.
- **Explain the event-driven paradigm** at an intuitive level, contrasting the reactive, "wait-then-respond" flow of an event-driven program with the top-to-bottom flow of a sequential program.
- **Reason about many-to-one triggering**, recognizing that a single device (or handler) may be activated by more than one distinct event.

10.3.1.2 Implementation Details

Logistics. The activity is unplugged and requires only inexpensive, readily available materials:

- One sheet of blank paper per team, used for scratch work, brainstorming, and the final labeled drawing of the machine.
- Writing implements (pens, pencils, or markers); colored markers are helpful but not required.
- A projector or slides to display the prompt, the worked example, and the reflection questions.

Students work in teams of three to four. The activity fits comfortably in a single class meeting and typically takes 30–35 minutes. The activity also scales well to large classes, since no equipment beyond paper is needed and teams work independently.

How to conduct. The activity proceeds in three phases: design, peer exchange, and reflection.

1. **Frame the task with an example.** Begin by showing an image of an "overly complicated machine" in which one action triggers the next. A sample image is shown in Figure 10.1.
2. **Design phase (teams of 3–4).** Ask students to form teams of three to four. One volunteer per team takes the sheet of paper for scratch work and brainstorming. As a team, students design an overly complicated machine with exactly five events that accomplishes some final goal. The machine must be drawn on the paper with each of the five events clearly labeled. Have each team member write their name on the back of the page (useful for participation credit and for keeping teams accountable).
3. **Peer-exchange phase (groups of 6–8).** Once every team has a design, ask each team to pair up with one other team, forming a larger group of six to eight students. Teams take turns describing their overly complicated machine to the other team, walking through the five events in order. Explaining the chain aloud forces students to articulate what triggers each step and what happens in response, which is precisely the event/response structure the lesson will name.
4. **Guided reflection.** Reconvene the class and lead a short discussion using the following prompts:
    - What was the most difficult part of designing an overly complicated machine with five events?
    - What if one of the devices had to be triggered by two events instead of just one? Would that be possible?
    - How does this activity mimic the way events happen in code? Use this question to pivot into the formal content.
5. **Transition to formal content.** With the shared experience in place, introduce the programming vocabulary, anchoring each term in what the students just did.

Instructional material. The activity is supported by a short lecture slide deck that (i) presents the prompt and a worked visual example of an overly complicated machine, (ii) states the step-by-step instructions for the design and peer-exchange phases, (iii) poses the reflection questions, and (iv) introduces the formal definitions of *event*, *response*, and *event handler*. Supplementary materials for implementing this activity (slides and handouts) can be found at USI's GitHub page for Unplugged Overly Complicated Machine.

10.3.1.3 Experience and Teaching Tips

The activity works best as a primer delivered immediately before the formal event-handling material rather than as a review. Students who have not yet seen event handlers

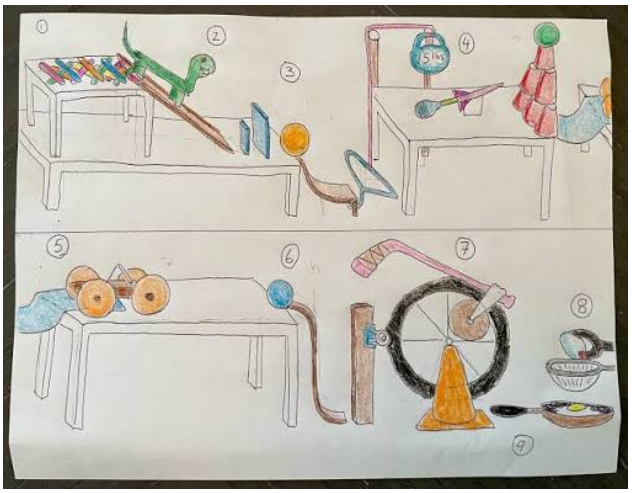


Figure 10.1: A sample "overly complicated machine." Each stage triggers the next, mirroring how one event leads to a response in an event-driven program. Image credit: the drawing is a replica of the Rube Goldberg machine created by a student in a CS2 course at USI in Spring 2026.

approach the design task without preconceptions and are then primed to attach the new vocabulary to a concrete, shared experience. The phrase "remember your machine" becomes a durable anchor for the terms *event* and *handler* throughout the unit.

Constraining teams to exactly five events matters. Too few and the chain is too trivial to reveal the idea of triggering; too many and teams spend the whole period negotiating an elaborate contraption instead of reflecting on structure. Five is enough to force students to think about how each step reliably sets off the next event.

Emphasize repeatedly that the drawing "can be ugly." Some students, especially those anxious about art, stall on making the picture look good. Framing the sketch as a communication tool that is good enough to explain to another team keeps the focus on the events and their ordering rather than on presentation.

The peer-exchange phase carries much of the learning. Instructors are advised to circulate during this phase and listen for teams that describe steps in cause-and-effect language, as these are ready-made examples to reuse when introducing handlers. Moreover, students do not automatically see that their paper machine is a model of software behavior. Instructors should state the mapping directly, explaining how each labeled step is an event, each next action is a response, and the "code" that would make the response happen is an event handler.

10.3.1.4 Example Implementations

The following institution(s) are using a variation of this assignment and you can find their details in their specific chapters:

- USI - CS2 Course Section §13.3.1
- TNTECH - CS2 Course TNTECH GitHub CS2 Overly Complicated Machine

10.3.1.5 Data and Analysis

Because the activity is unplugged and self-contained, it lends itself to a range of lightweight assessments that any instructor can adopt, from purely observational evidence to more structured instruments. Instructors can select among the following suggested options according to how rigorously they wish to evaluate learning and how much class time they can devote to measurement.

- **Artifact-based assessment with a rubric.** The labeled drawing each team produces is a ready-made, gradable artifact. A simple rubric can score whether the machine contains exactly five distinguishable events, whether each event is clearly labeled, whether the triggering relationship between consecutive steps is explicit, and whether the chain accomplishes a coherent final goal. Scoring the sheets provides direct evidence of whether students can identify and sequence discrete events.
- **Paired pre-/post concept questions.** A short set of conceptual questions, administered immediately before and after the activity (or before the unit and after it), can measure gains against the stated learning outcomes: defining an event and an event handler, distinguishing an event from its response, contrasting event-driven with sequential control flow, and reasoning about a single handler responding to more than one event. Keeping the same items on both administrations allows a straightforward pre/post comparison.
- **Embedded exam or quiz questions.** One or two questions on a later quiz or exam that ask students to recall the event/response/handler mapping (optionally referencing "your machine") give a delayed measure of retention and let instructors track the concept alongside the rest of the course.
- **Analysis of reflection responses.** The guided-reflection prompts can be collected as short written responses (for example, an exit ticket or minute paper) and coded qualitatively for whether students articulate cause-and-effect triggering, recognize the possibility of many-to-one triggering, and connect the physical machine to program behavior.
- **Transfer assessment via a follow-on coding task.** If the activity is paired with a plugged-in assignment, the graded coding artifact serves as an assessment of transfer to check whether students can turn an event chain into working code with listeners and handlers. Chapter 13, describes this assessment methodology used in a CS2 course at USI.

Even in the absence of a formal instrument, instructor observation during class remains a valuable formative signal. Monitoring engagement while teams brainstorm and draw, listening to the explanations exchanged during peer sharing, and attending to the depth of responses during the guided reflection all indicate whether students can articulate their five events, reason about triggering a single device with two events, and connect their physical machine to the notions of events and event handlers. Such observational evidence is formative and anecdotal rather than the result of a controlled study, but combined with one or more of the assessments above it gives instructors a practical, low-overhead way to gauge whether the activity meets its intended learning outcomes.

10.4 New Plugged-in Activities

10.4.1 Greenfoot Event Handling Activity

Greenfoot is an interactive, visual software environment designed to help beginners learn object-oriented programming using the Java language [34]. It lets users build 2D games, simulations, and graphical programs. Instructors at Knox College utilized Greenfoot interface for teaching event handling concepts that rely on a poll-driven (or continuous check) model inside an `act()` method. In this section, a description of the activity and how it can be used to teach event handling in a CS2 class is reported. Moreover, the conclusion of this section provides links to individual chapters detailing example implementations at other universities.

This activity is essentially the same as the CS1-oriented Greenfoot activity discussed in §1.4.2. The activity is included in both locations because it was adopted at both the CS1 and CS2 levels.

10.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

This plugged-in activity asks student teams to design and build an original two-dimensional game in Greenfoot whose core mechanics are driven entirely by events. The interface provides an environment in which a `World` is populated by `Actor` objects. Rather than writing an explicit main loop, a student overrides an `act()` method on each actor, and the environment invokes that method repeatedly on every step of a simulation loop. Program behavior therefore emerges from how each actor responds to events such as key presses, mouse clicks, timer ticks, and collisions with other actors, while the framework, not the student, decides when each handler runs. This inversion of control is the central idea of event-driven programming, and it is what makes Greenfoot a natural vehicle for teaching it.

The activity is deliberately open-ended. Students receive a small set of runnable example projects, ideally one minimal and one feature-rich, that demonstrate movement, simple animation, enemies, and interaction between actors. They are then required to produce a game that is meaningfully different from the examples, through original mechanics, assets, or gameplay features, rather than a minor modification of a sample. This requirement pushes students beyond copying working code and toward deciding for themselves which events their game must respond to and how those events should be handled.

Conceptually, every team ends up implementing some variation of the following per-step cycle for each actor:

1. Read input and detect events (is a key held down or was it pressed this step? was the mouse clicked or moved?).
2. For each detected event, run the corresponding handler (move, fire, change the sprite, play a sound).
3. Detect collisions with other actors and run the collision handler (update the score, remove an actor, end the round).
4. Advance time- or state-based behavior (enemy movement, animation frames, spawn and restart timers).

5. Update the world state and the score.

Prerequisites. Students need working knowledge of Java syntax, classes, objects, methods, instance variables, and loops. Knowledge of inheritance is helpful, but the use of subclasses in Greenfoot is simple enough that students who have not yet studied it formally can still complete the activity by following the examples. Students with a background in another object-oriented language (for example C++) have been reported to pick up enough Java to use Greenfoot for one or two assignments. No prior game-development or Greenfoot experience is assumed. Because the activity concentrates on the event-handling concept, it works best when students have first been introduced to events in a lecture and, optionally, through an unplugged exercise in which they model a system as a set of discrete event–response pairs.

Learning outcomes. By completing the activity, students should be able to:

1. Identify the distinct events a program must respond to and map each event to a handler;
2. Implement keyboard-, mouse-, timer-, and collision-driven event handling across multiple interacting actors using the Greenfoot API (e.g., `Greenfoot.isKeyDown()`, `Greenfoot.getKey()`, `Greenfoot.mouseClicked()`, `getOneIntersectingObject()`, and the per-step `act()` contract);
3. Reason about a program whose control flow is dispatched by the framework rather than written as an explicit sequential loop, including the fact that handler ordering is undetermined unless the programmer explicitly specifies it (e.g., with `World.setActOrder()`), which can otherwise lead to unpredictable results;
4. Integrate and manage external assets (custom sprites, images, and sounds) through the project's resource folders and the appropriate constructor code; and
5. Explain and demonstrate the non-trivial event-handling features of their own game to others.

10.4.1.2 Implementation Details

Logistics. The activity is designed as a team project, with teams of two to four students. Teams should be small enough that every member writes some of the event-handling code, rather than a single member owning all of the logic. However, the activity can be implemented as an individual assignment as well. Greenfoot is free and runs on Windows, macOS, and Linux, so no special lab infrastructure is required. Students can use their own laptops or lab machines, and installation should be completed before the first working session. Activity implementation usually requires one class period for the event-handling lecture (plus one for an unplugged activity, if used), one lab period for the worked examples, and one to two weeks of project time with some in-class work sessions.

Games depend on artwork and sound, so instructors should decide up front how assets may be obtained. A convenient policy is to permit free asset packs from public sites (such as `itch.io`), require students to credit every third-party asset, and require each team to keep a record of each asset's source link and license. This doubles as a lesson in professional practice. Finally, instructors should choose the deliverables (the working game and source,

a short technical write-up, and optionally a demonstration video), the submission channel (e.g., a learning management system), and the grading rubric, and share them with the students when the project is assigned.

How To Conduct. The activity is most effective when it is sequenced from concept to code rather than opening directly with the IDE:

1. **Introduce events.** Give a short lecture on event-driven programming that presents an event as a discrete point to which a program responds, and contrasts framework-dispatched control flow with a sequential main loop.
2. **Study worked examples.** Distribute the runnable example projects, and have students run, read, and modify them. A minimal example gives every team an entry point, while a feature-rich example provides a ceiling for stronger teams. Prompt students to locate, for each behavior they see on screen, the event that triggers it and the handler that implements it.
3. **Design and build.** Teams brainstorm a game, list the events it needs, agree on a minimum set of features, and then implement, test, and refine iteratively. Instructors circulate during in-class work sessions and ask teams to explain which event triggers a given piece of code.
4. **Reflect and share.** Teams submit the game together with a short write-up describing a few features they found interesting or technically challenging, and why the event handling for each was non-trivial.

Instructional material. The activity is supported by the following materials, which can be adapted to any course:

- Lecture slides introducing event-driven programming, events as discrete response points, and the Greenfoot execution model (the `act()` contract, the world and actor classes, and the relevant parts of the Greenfoot API).
- Worked example projects, distributed as runnable Greenfoot projects, ranging from a minimal example to a richer one that includes movement between screens, simple animation, enemies, and dialog windows. A short walkthrough video by the example's author describing its structure, including how custom images are stored and loaded, greatly reduces setup and asset-loading questions.
- How-to guides covering installation (download from `greenfoot.org`), using custom images (placing files in the project's `images` folder and loading them in an actor's constructor with a short provided code snippet), finding and crediting third-party assets, and submitting deliverables.
- Assignment specification and rubric that define the task, minimum requirements, deliverables, submission process, and a grading rubric for the event-driven implementation. In place of a conventional test, assessment is performance-based, on the working game and the team write-up.

Materials for this activity, including background material, assignments, videos, and example code, are available in a shared GitHub repository: Greenfoot Teaching Material.

10.4.1.3 Experience and Teaching Tips

- **Preserve the concept-first ordering.** Instructors who moved from an unplugged event-modeling exercise to the Greenfoot project found that teams arrived already thinking in terms of discrete events and handlers, which made the leap from a diagram to code much easier.
- **Provide two contrasting worked examples.** A minimal example gives weaker teams a base to follow, and a feature-rich one gives stronger teams a ceiling to reach past. Pairing the richer example with a creator's walkthrough video substantially reduces setup and asset-loading questions.
- **Make the assignment open-ended, with a minimum but no maximum.** Specify a minimum set of requirements (for example, at least one keyboard event, one mouse event, one collision, and one timer- or state-based behavior), but do not cap what teams may do. Trusting students' creativity encourages ownership, and many teams go well beyond the examples.
- **Require original games.** State that the submission must be meaningfully different from the samples, and disallow games that already have many public implementations.
- **Keep teams small.** Ensure that every member touches the event-handling code; consider requiring a brief contribution statement.
- **Require a short technical write-up.** Asking teams to describe a few features that were technically challenging, and why, pushes students to articulate the event-handling logic in their own words and consolidates the learning.
- **Budget in-class time.** Students most often ask for more in-class time to work on the project and to become familiar with the environment before starting. A short guided lab between the worked examples and the open project is a low-cost way to address this.

10.4.1.4 Example Implementations

The following institution(s) are using a variation of this assignment and you can find their details in their specific chapters:

- Knox - CS1 Course Section §3.3.3
- USI - CS2 Course Section §13.4.1
- Casper - CS2 Course Section §17.4.3
- MSU - CS2 Course Section §18.3.1

10.4.1.5 Data and Analysis

Rather than reporting results from a particular offering, this section suggests ways for instructors to assess whether the activity is helping students learn the parallel concept of event handling. No single measure is sufficient. We recommend triangulating across at least one direct measure of the artifact (what students built), one direct measure of understanding (what students can explain or predict), and one indirect measure (what students say about their experience). Classes for this activity are often small, so descriptive and qualitative summaries are usually more appropriate than inferential statistics. Moreover, instructors

interested in publishing findings should check their institution's requirements for human-subjects review before collecting survey or artifact data. The chapters listed in the subsection above provide a more detailed analysis of the specific implementations reported by other university instructors.

10.4.2 Fork-Join Activity

This "activity" is a unit of the class, consisting of approximately 100 minutes of lecture, some lab exercises, and related assignments.

10.4.2.1 Activity Summary

The goal of this activity is to introduce students to the fork-join framework in Java, which facilitates the use of threads to complete recursive calls. Behind the scenes, it creates a pool of threads that work on available tasks. When a recursive call is made, it is stored as a task until a thread is available to run it. The general philosophy is to create significantly more tasks than threads (called *overdecomposition*), which provides automatic load balancing (threads which finish tasks quickly naturally do more of them), but it is also important to limit overhead by providing a cutoff value so that tasks below a certain size solve the problem themselves rather than creating further recursive tasks.

We believe an important feature of this activity is the use of simplified notation. Using the fork-join framework in Java requires creating a class whose objects are the tasks and invoking long-winded library calls such as `ForkJoinPool.commonPool().invoke`. Instead, the lecture teaches the bulk of this material using two new keywords `fork` and `join`, following their use in the algorithms community, including in the common textbook CLRS [13]. Java code written with these keywords will not compile, but their use at the beginning of the activity allows students to focus on the parallel computing content without distraction from the notation. This is helpful since the students have limited prior exposure objects representing something other than data or to chaining method calls as in `ForkJoinPool.commonPool().invoke`. The true notation is shown at the end of the lecture so that students are prepared to use it in lab and programming assignments.

The idea of teaching the fork-join framework in a data structures course was previously explored by Grossman [27, 28] and this part of our materials uses ideas from the materials he created to teach this material [26]. This activity differs in two ways. One is the notation; their materials consistently use the actual Java notation, which we found challenging for our students. The other is what else is taught in addition to the fork-join framework. Our activity spends time practicing the idea of dependencies and possible execution orderings. Instead, their materials teach the distinction between parallelism and concurrency (using more resources to do more operations at once and ensuring correctness when multiple threads use shared resources), more advanced algorithms and analysis techniques, and more advanced primitives like condition variables. This is reflected in the amount of time spent (100 minutes of lecture for our activity vs 8–10 hours of instruction for theirs). Thus, our activity is easier to fit into an existing course, while theirs covers more advanced content.

10.4.2.2 Prerequisites

The main prerequisites needed for this activity are recursion and object-oriented (OO) programming. Recursion is important because the framework is intended to work on recursive functions so the students need to be able to write a recursive routine for whatever they will be asked to do. Alternately, the instructor could provide the recursive routine; this leaves the student with the relatively-mechanical part of adding the framework, though students could be asked to study the effect of changing the cutoff value in this case. Ideally, the students are able to compute the depth and total work of the recursion tree for their program, but this unit has been used at Knox College with students who don't know how to solve recurrences yet by walking them through the solution of the specific recurrence of interest.

The other main prerequisite is familiarity with OO programming in Java. Using the fork-join framework requires creating a class that extends `RecursiveTask` and implements its `compute` method to complete the task. Since the `compute` method doesn't take arguments, there must be a constructor that takes information required for the computation and stores it in instance variables. The students do not need a high level of experience since they can follow the given example code, but some familiarity is important because the students are likely also confronting the new idea that objects can represent bundles of work and not just data.

The activity does not assume prior exposure to parallel computing. One of the first slides introduces the idea of multiple cores. In our experience, the students are aware of this terminology, but not necessarily what it means. The instructor can adjust the activity based on the level of student familiarity by spending more or less time on this slide, such as explaining cores are separate workers that can perform separate operations simultaneously if the students don't have prior exposure to the idea of parallelism.

10.4.2.3 Learning Outcomes

Upon completing the activity, students will be able to do the following:

- Write parallel programs using Java's fork-join framework,
- Justify the use of overdecomposition and a cutoff value to end recursion,
- Define the work and span of a directed acyclic graph and explain their importance, and
- Define the parallel concept reduction, give some examples, and implement them in Java using the fork-join framework.

10.4.2.4 Logistics

The provided slides represent approximately 100 minutes of lecture. They are designed to work with the Peer Instruction [47, 49] pedagogy, in which lectures are broken up with multiple choice questions. The students answer these individually, discuss their answers in small groups, and then answer again. Thus, the slides have periodic questions. Instructors not using this pedagogy could replace the questions with worked examples.

The activity also contains two lab assignments, plus an alternative which has not been tried in class in its current form. These include a handout and given code, with the idea that

the students follow directions on the handout, possibly asking questions of the instructor during this process. In our experience, most students can complete the labs within a 70-minute period. The first lab assignment should be given after most of the lecture material is taught. (The students need up to the part of the slides showing the real Java code, which is late in the slide deck so that students can learn the material with the simplified notation.)

Different ways to teach this material are discussed in §10.4.2.8.

10.4.2.5 Running the Activity

Now we will go through the main parts of the activity.

Lecture. This section talks through the main parts of the lecture. Slide by slide detail of the points to make on each slide are in the notes of the Powerpoint version of the slides.

The lecture begins by introducing the concept of multiple cores working in parallel and then demonstrates the `fork` and `join` keywords in the context of summing the values in an array:

```
int sum(int[] array, int low, int hi) {
     int left = fork simpleSum(array, low, (hi+low)/2);
     int right = simpleSum(array, (hi+low)/2, hi);
     join;
     return left + right;
}
```

where `simpleSum` is a method that uses a for loop to sum the values of `array` from `low` up to (but not including) `hi`. The `fork` keyword allows the first call to `simpleSum` to run in parallel with the rest of the method. The `join` keyword causes execution to wait until all calls to `fork` have finished, ensuring that both `left` and `right` have the correct values before they are added. To practice these, we look at the orders in which short code segments such as the following could print:

```
System.out.print(1);
fork System.out.print(2);
System.out.print(3);
System.out.print(4);
join;
```

(1 must print first, and 3 must print before 4, but 2 can be anywhere relative to 3 and 4.)

Next we introduce precedence graphs, directed acyclic graphs (DAGs) which shown which operations must run before others. In such a graph, the tasks are boxes (each called a *node*) and directed edges indicate when one node must run before another. Figure 10.2 shows a precedence graph and the arrow from a to b means that a must complete before b can start. A precedence graph can be constructed from a block of fork-join code and the graph makes it easier to define our metrics of running time.

For a serial program, the precedence graph would be a linear chain of nodes and its running time would be the sum of those running times. Computing the running time for a

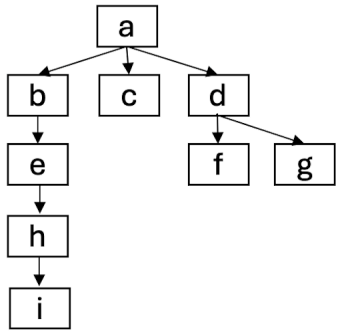


Figure 10.2: Precedence diagram of tasks a–i.

parallel program is more complicated because pairs of tasks that are independent (i.e. don't have paths from one to the other) can be run simultaneously. There are two metrics of interest: *span*, the length of the longest path, and *work*, the total running time of all the nodes. For simplicity in going from precedence graphs to running time, we typically assume that each node takes 1 unit of time to complete. We also assume that each core will run a node if it can, i.e. it won't be voluntarily idle. With these assumptions, the DAG in Figure 10.2 can run in 5, 6, or 7 time units on 2 cores, depending on the schedule. Here are schedules achieving each of these:

a	b	e	h	i
	c	d	f	g

a	c	b	e	h	i
	b	f	g		

a	c	f	b	e	h	i
	d	g				

This specific graph also highlights the importance of span; even with more cores, the processing time cannot be less than 5 since this is the length of the longest path. The students also see other examples where the work is the main factor in the processing time. Together, these motivate the idea that we're looking to minimize both work and span when considering multi-threaded algorithms.

Next we show the reason for overdecomposition using recursion (and using `fork` to allow a recursive call to run in parallel). The alternative is to split the problem into a set number of pieces (such as the number of cores) and run those in parallel. Using a specific number of pieces assumes that we know the number of available cores, which could be incorrect if some of the cores are busy with another program. It also assumes that we can evenly split the work into tasks, which may not be true if the task has a complicated structure. (Even a loop is tricky if the iterations have unpredictable amounts of work.) Thus, it can be advantageous to split into a larger number of tasks, giving the scheduler flexibility to achieve good load balance. This part of the lecture completes the conceptual version of the fork-join code, shown in the left side of Figure 10.3.

The main part of the slides concludes by introducing the actual notation for the fork-join framework, essentially moving from the left side of Figure 10.3 to the right side. The subproblems are represented with objects of type `Sum` and the notation for solving them reflects this. A threshold is also introduced so the code only recurses on problems above a certain size to limit overhead.

```
int sum(int[] array, int low, int hi) {
    if(hi-low <= 1)
        return simpleSum(array,low,hi);

    int mid = (hi+low)/2;
    int left = fork sum(array,low,mid);
    int right = sum(array,mid,hi);
    join;
    return left + right;
}

//in main:
val = sum(array, 0, array.length);
```

```
class Sum extends RecursiveTask<Integer> {
    //constructor and attributes

    protected Integer compute() {
        if(hi-low <= THRESHOLD)
            return simpleSum(array,low,hi);

        int mid = (hi+low)/2;
        Sum left = new Sum(array,low,mid);
        Sum right = new Sum(array,mid,hi);
        left.fork();
        return right.compute() + left.join();
    }

//in main:
Sum obj = new Sum(array,0,array.length);
val = ForkJoinPool.commonPool().invoke(obj);
```

Figure 10.3: Fork-join code to sum values in an array. Left: Version with fictional `fork` and `join` keywords. Right: Authentic Java version.

The last couple of slides introduce generalizations. The first explores the idea of *reductions*, which take an array and returns a single value derived from it. Sum is one reduction, but the slide lists a bunch of others, including product, minimum value, maximum value, and creating a histogram. The idea is to point out these different problems and realize that they can all be implemented with very minor changes to the code. The other generalization is to higher dimensions; very similar code would also sum the values of a 2D array.

Lab exercises. Once students have the background knowledge, it is time for them to apply it. That is the role of our lab exercises; these include a handout that talks the student through tasks that review and build on the material from lecture. These are intended to be done with the instructor available to answer questions and before homework assignments are given. (The labs tasks can also be given as homework, but this is not the original intent.) In terms of timing, the lab assignments should be given after the main part of the slides, i.e. after the students have seen how to implement recursive fork-join code in real Java.

The first lab (called Lab 8 because that was it's position in the course it was created for) asks the student to implement a fork-join program to count prime numbers within a range. This is a slight generalization of summing an array; you can think of summing an array of 1s and 0s where the value at an index is determined by whether that number is prime or not. A feature of this problem is that splitting the range of indices evenly yields unequal work because it takes longer to check larger numbers for primality. (This property has been observed before [11].) In our experience, most students are able to complete this lab within a 70-minute lab period.

The second lab (called Lab 9) asks students to explore the idea of reductions. First, they implement finding the minimum and finding the kth smallest. This illustrates how little code has to change to implement different reductions. Then, the students are asked to implement a *scan* operation, one which creates a new array such that the value at index i is the result of the reduction on the first i input values. This requires a second recursive call for each subproblem, but do not change the asymptotic work or span. (A particularly nice treatment of scan can be found in Lin and Snyder [38].) This lab does not require a large amount of code to be written/modified, but it is conceptually dense so many students needed additional help.

In addition to the two labs assigned in versions of this class, the repository contains another lab that has not be class tested as a fork-join problem. For this lab, the students are provided with a working serial simulation of heat diffusion and they are asked to parallelize it. The given code is significantly larger, though the students are given a hint about where to look. Parallelizing this problem has been assigned in the past [25, 43], but not with the fork-join framework.

Homework. Normally, we believe it is important to follow up lab assignments with homework assignments on the same material. This assignment has been somewhat of an exception to this policy because it occurs in the final weeks of our term. The one assignment that was given was to complete the 2D generalization discussed in the first lab (modify the code to sum the values in a 2D array). The majority but not all students were able to complete this assignment. Other natural assignments would be to implement a reduction or scan.

10.4.2.6 Instructional Material

All materials needed to run the activity are in the Knox College CS2 chapter's part of the GitHub repository. The repository contains slides, lab handouts and assignments

Instructors thinking about adopting this material can also refer to two recorded lectures on this material (1, 2). The lectures use a slightly older version of the slides, but the changes are relatively minor. These videos are not intended for dissemination to students, but are provided for instructor reference.

10.4.2.7 Example Implementations

Another resource useful to instructors interested in adopting this material is the accounts and materials of project members who have done so. This activity has been implemented at the following institutions:

- The materials were created for use in CS2 at Knox College. That implementation is discussed in §12.3.
- The activity was implemented at the University of Southern Indiana. That implementation is described in §13.4.2. Notably, this activity was combined with an unplugged fork-join activity, described in §13.3.2.
- A shortened version of the activity was also implemented at Casper College. This version focused on precedence graphs and the span and work metrics, having the students determine the running time of given graphs. It is described in §17.3.3.

The reader is referred to the linked chapters for more details about each implementation.

10.4.2.8 Experience and Teaching Tips

Our main lesson from teaching this activity is that using the fork-join keywords initially is very helpful. Students get intimidated by the real Java notation and this makes it harder for them to learn the concepts. Student difficulties also led to to add slides checking understanding of the concepts: do they understand the flexibility that `fork` provides? Can they find long and short schedules that are allowed by a given graph? Can they calculate work and span? At first this felt unnecessary, but answering questions on these makes the concepts more tangible for students.

Numerous variations of this activity are possible. The implementation at Casper College (§17.3.3) removes the programming elements to focus on the conceptual material. The second lab and the material on reductions can definitely be removed from the end. Alternately, a longer version of the activity could be constructed by adding more material from the notes of Grossman [26]. In addition, either of the labs could also be replaced by the one that uses heat diffusion or other interesting applications of this material could be developed into labs. Section §12.5 talks about planned changes in the course including this activity.

10.4.3 Plugged-In Sorting Algorithm Comparison

10.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

This assignment introduces parallel programming in CS2 by extending a topic that students are already studying: sorting algorithms. Students compare sequential and parallel implementations of familiar sorting algorithms and experimentally investigate when parallelization improves performance and when it does not.

Students work with four primary sorting implementations:

- sequential Bubble Sort,
- parallel Odd-Even Transposition Sort,
- sequential Merge Sort, and
- parallel Merge Sort.

Students also compare these implementations with the GNU parallel sorting library. Each algorithm is executed on both integer and string datasets so that students can observe whether algorithm and parallelization behavior changes with the type of data being processed. The assignment provides small datasets for correctness testing and substantially larger datasets for performance experiments. The provided assignment uses large files containing approximately 31,100 strings and 37,084 integers.

An important feature of the assignment is that students do more than implement the algorithms. After completing the program, they conduct a performance experiment in which they vary the number of OpenMP threads, record execution times, compare sequential and parallel algorithms, and explain the observed results. In particular, students investigate the counterintuitive result that adding more threads does not necessarily make a program faster. The assignment deliberately includes experiments with 4, 20, and as many as 500 requested threads to make parallel overhead visible.

This structure allows parallel and distributed computing concepts to be incorporated without removing existing CS2 content. Students still practice sorting, recursion, templates, vectors, algorithm analysis, and testing, but the familiar problems become a vehicle for introducing concurrency, shared-memory parallelism, synchronization, thread overhead, and scalability.

Before beginning the assignment, students should be comfortable with the following CS2 concepts:

- C++ functions and parameter passing,
- vectors and indexed access,
- C++ templates,
- file input,
- iterative sorting algorithms,
- Merge Sort,
- recursion and recursive call trees,
- basic algorithmic efficiency,
- testing and debugging programs incrementally, and
- measuring or interpreting program execution time.

Students do not need previous parallel programming experience. OpenMP is introduced in lecture and example code shown prior students being assigned this program. The parallel constructs required are intentionally limited to a small subset of OpenMP:

- `omp_set_num_threads()`,
- `#pragma omp parallel`,
- `#pragma omp for`,
- OpenMP reductions,
- `#pragma omp parallel sections`, and
- `#pragma omp section`.

The assignment therefore works particularly well after students have already learned Bubble Sort and Merge Sort. Parallelization becomes an extension of algorithms they understand rather than an additional algorithmic problem layered on top of unfamiliar material.

After completing this assignment, students should be able to:

- Implement sequential sorting algorithms using both iterative and recursive approaches.
- Explain why some algorithms are more naturally parallelizable than others.
- Use OpenMP directives to parallelize independent computations in a shared-memory C++ program.
- Identify independent units of work within an algorithm that may safely execute concurrently.
- Use an OpenMP reduction to safely combine results produced by multiple threads.
- Explain the purpose of synchronization in parallel algorithms and recognize situations in which unrestricted concurrent access could produce a race condition.
- Compare sequential and parallel implementations empirically using measured execution times.
- Explain why increasing the number of threads does not necessarily decrease execution

time.

- Describe the relationship among the number of threads, available processors, computational work, and parallel overhead.
- Recognize that algorithm selection can have a greater effect on performance than parallelization.
- Limit parallel recursion to avoid excessive thread creation and parallel overhead.
- Evaluate parallel performance experimentally rather than assuming that a parallel implementation will be faster.

Students must reason about where parallelism exists, how the implementation changes the behavior of an algorithm, and whether the resulting parallel program actually improves performance.

10.4.3.2 Algorithms

The assignment was intentionally designed around two sorting algorithms with substantially different characteristics: Bubble Sort and Merge Sort. This contrast provides students with an opportunity to examine both algorithmic efficiency and parallel scalability.

Sequential Bubble Sort. Students first implement a conventional Bubble Sort. Adjacent elements are compared and exchanged when they are out of order. Multiple passes are performed until the vector is sorted. The implementation includes an early-termination optimization. A Boolean variable records whether a swap occurred during a pass. If an entire pass completes without a swap, the algorithm terminates because the vector is already sorted. Bubble Sort is useful pedagogically because its `(O(`n^2`))` behavior becomes clearly visible on the large datasets used for the assignment. This gives students a baseline against which the other algorithms can be compared.

Parallel Odd-Even Transposition Sort. A traditional Bubble Sort cannot simply have its inner comparison loop divided among threads because adjacent comparisons may depend upon and modify overlapping elements. The parallel version therefore uses Odd-Even Transposition Sort, sometimes called Brick Sort. The assignment describes it to students as a parallel variation of Bubble Sort because elements continue to move toward their sorted positions through repeated adjacent compare-and-swap operations. The algorithm alternates between two phases. During the even phase, the following independent pairs are compared: (0,1), (2,3), (4,5), etc. During the odd phase, the pairs are: (1,2), (3,4), (5,6), etc. Within an individual phase, no two comparisons modify the same vector element. The comparisons can therefore be distributed across multiple threads. This provides an especially useful introductory example of data parallelism. Students can visually identify which operations are independent and consequently understand why those operations may safely execute simultaneously. A shared Boolean variable, `swapped`, determines whether another iteration is necessary. Because multiple threads could discover that a swap is required, the assignment introduces the OpenMP logical-OR reduction: `reduction(||:swapped)`. Each thread receives a private version of swapped, and OpenMP combines those values using logical OR after the work is completed. Thus, if any thread performs a swap, the final value is true. The assignment explicitly connects the reduction operation to avoiding race conditions and unnecessary synchronization on a shared Boolean variable.

Sequential Merge Sort. Students also implement recursive Merge Sort. Each recursive call divides a vector range into two halves: (1) recursively sort the left half, (2) recursively sort the right half, and (3) merge the two sorted halves. A shared `merge()` helper function combines the two ordered ranges into a temporary vector and copies the result back to the original vector. Merge Sort provides an important contrast with Bubble Sort because its $O(n \log(n))$ complexity makes it dramatically faster for large datasets even before parallelism is introduced.

Parallel Merge Sort. Merge Sort also contains an obvious source of task parallelism. Once the midpoint has been determined, sorting the left half and sorting the right half are independent operations. The parallel version places those two recursive calls into separate OpenMP sections: one section recursively sorts the left half, another recursively sorts the right half. The assignment therefore allows students to encounter task parallelism after seeing data parallelism in the Odd-Even Sort. Unrestricted recursive parallelization, however, would continually create additional parallel work as the recursion tree expands. The assignment addresses this with a depth parameter. Parallel sections are created only while the recursion depth is less than four. At deeper levels, the same recursive algorithm continues sequentially. This design decision introduces the concept that creating parallelism has a cost. A correct parallel program is not necessarily an efficient parallel program. The depth restriction also provides a concrete example of granularity control. Parallelism is used for relatively large subproblems near the top of the recursion tree, while smaller subproblems are processed sequentially.

10.4.3.3 Implementation Details

The assignment is implemented in C++ using OpenMP and the GNU parallel algorithms library. Students are provided with a partially completed driver program so that the assignment remains focused on algorithms and parallelism rather than file-processing boilerplate. The provided code includes the required libraries, portions of `main()`, file-reading functionality, a vector-printing function, and a scrambling function. Students complete the sorting functions, timing code, input validation, and function prototypes.

All sorting functions are implemented as C++ template functions. Using templates serves two purposes. It reinforces an existing CS2 concept while also demonstrating that the parallelization strategy is based on the structure of the algorithm rather than a particular primitive data type. The same implementations are tested with: `vector<int>`, and `vector<string>`.

The program asks the student to specify the number of threads OpenMP should use. Input validation requires an integer from 1 through 500. Dynamic adjustment of the thread count is disabled, and the requested thread count is supplied to OpenMP. The program also reports the number of logical processors indicated by the computer. This enables an important experimental distinction between the number of threads requested by the program, and the amount of hardware capable of executing those threads simultaneously. Students can then intentionally request more threads than available hardware contexts and observe the resulting behavior.

Execution time is measured with C++'s `chrono::high_resolution_clock`. The general experimental pattern is:

- scramble the vector,
- record the starting time,
- execute one sorting algorithm,
- record the ending time, and
- calculate and display elapsed seconds.

The vectors are scrambled again before the next algorithm is measured so that each algorithm receives unsorted data. Students measure integer and string performance separately for every sorting implementation. On systems using GCC, the program is compiled with OpenMP enabled using the `-fopenmp` option.

The original assignment also provides separate setup instructions for macOS because Apple's default Clang configuration does not provide the same out-of-the-box OpenMP support. Students using macOS are directed to install GCC through Homebrew and compile using the installed GCC version.

10.4.3.4 Instructional Material

All materials including the supporting lecture materials, assignment document, and files given to students are in the TNETECH CS2 chapter's part of the GitHub repository at Plugged-In Sorting Algorithm Comparison.

10.4.3.5 Example Implementations

The following institutions are using a variation of this assignment and you can find their details in their specific chapters:

- TNTECH - CS2 course - Section 11.3.1
- UNL - CS2 course Section 14.4.2
- Casper College - CS2 Course Section 17.4.4

10.4.4 Parallel Image Processing

Activity Summary

The parallel image design is meant for higher-education CS departments that want an instructor-ready activity with both conceptual scaffolding and implementation detail [68]. The plan follows the lecture-demo-lab structure, uses the provided Java multithreading slide deck as the conceptual guide, and translates the parallel image blur lab into a fully scaffolded CS2 lesson. Pedagogically, the strongest reason to use image processing for CS2 concurrency is that it is both inherently data parallel and visually self-explanatory. It also recommends a combined sequence of lecture, demonstration, and experiment rather than lecture alone. The provided slide deck introduces parallelism, concurrency, race conditions, synchronization, speedup, overhead, `ExecutorService`, and mathematics for various image-processing kernels.

Students begin with a short unplugged partitioning warm-up inspired by FlagMaker in CS1, a course they took in the previous semester, then a conceptual lecture to present the terminologies in PDC, code explanation on edge detection and histogram generation with

thread synchronization. Then, students are asked to implement two multi-threaded Java programs for box blur lab: a sequential baseline, a parallel row-partitioned version. They measure runtime across thread counts, compare outputs, calculate speedup by hand, and discuss why more threads do not always produce faster execution. The learning target is not merely "use threads," but to understand the trade-offs among correctness, overhead, scalability, and code complexity. For a default implementation, this plan assumes at least two class meetings totaling about 135 minutes: one concept-and-demo session and one guided lab session. If your department runs longer lecture, the same material can be collapsed into a single 115 minute workshop by shortening discussion and moving the debrief outside class.

10.4.4.1 Example Implementations

The following institutions are using a variation of this assignment and you can find their details in their specific chapters:

- MSU - CS2 course - Section §18.3.2
- USI - CS2 course Section §13.4.3

Student Learning Outcomes

By the end of this activity, students should be able to:

- Explain the difference between sequential execution, concurrency, and parallelism; use Java multithreading tools to implement a row-partitioned image kernel.
- Analyze the trade-off between speedup and overhead; distinguish embarrassingly parallel work from work requiring synchronization.
- Apply synchronization mechanisms when necessary.

Conceptual objectives for students are these:

- Distinguish parallelism from concurrency and describe the role of a thread versus a processor/core.
- Define race condition, critical section, synchronization, speedup, and parallel overhead.
- Predict why runtime improvement may plateau or reverse when thread count becomes too large.

Prerequisite

This lab is best placed in CS2, after students are comfortable with loops, arrays, methods, objects, file I/O, and basic exception handling. The Java lab assumes JDK installation and basic Java familiarity. A short pre-lab warm-up is therefore recommended: have students mark a 6×6 pixel grid and identify which cells one worker, two workers, and four workers would compute. This preserves the "students take the role of processors" spirit of Unplugged Flag Maker Activity while leading naturally into row partitioning for the lab. Students do not need a prior course in operating systems, architecture, or high-performance computing.

Algorithms

The Basic Idea: Convolution A grayscale image is a grid of numbers: $I(x, y)$ is the brightness of the pixel at column x, row y (0 = black, 255 = white).

Blurring operation slides a small 3×3 matrix K (called a kernel) over the image. At each pixel, multiply each kernel entry by the pixel under it and add everything up:

$$I'(x, y) \;=\; \sum_{i=-1}^{1} \sum_{j=-1}^{1} K(i, j)\; I(x + i,\; y + j). \tag{10.1}$$

Blurring replace each pixel with the **average of itself and its 8 neighbors**. In kernel form:

$$K_{\text{box}} = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{10.2}$$

Every neighbor counts equally. The weight of each pixel is the same, so the overall brightness of the image does not change. The blurred pixel is:

$$I'(x, y) = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} I(x + i,\; y + j). \tag{10.3}$$

Example. If the 9 pixels under the kernel are $10, 10, 10, 10, 10, 200, 10, 200, 200$, the new center value is their average:

$$\frac{10 + 10 + 10 + 10 + 10 + 200 + 10 + 200 + 200}{9} = \frac{660}{9} \approx 73.$$

A bigger kernel (e.g. 5×5 with $\frac{1}{25}$) averages over more neighbors and gives a stronger blur.

Materials and Running Environment

All teaching materials can be found in github repository with files listed in Table 10.3. The original blur lab specifies JDK 8+, `images/input/` and `images/output/` folders, and an example image named `aroma.jpg`. It also expects two Java files: a sequential baseline, and a parallel skeleton.

Instructor Notes and Teaching Tips

Note: For beginners, distribute instructor-curated images rather than asking students to search for their own during lab time. This keeps the lesson focused on multithreading rather than file wrangling and reduces licensing ambiguity. If students later use personal images for extension work, require them to note the source and confirm reuse permissions.

The most important misconception to surface early is that concurrency and parallelism are not the same thing. The slide deck distinguishes concurrency as multiple tasks making progress and parallelism as simultaneous execution on multiple cores. Students often overuse

Table 10.3: Instructor Resources.

Resource	Purpose
Lecture slides: `class10_PDC.key`	Introduce sequential vs. parallel thinking, overhead, speedup, race conditions, synchronization, and the image-processing example.
Student handout: adapted `CSIT112 Lab5_PDC.pdf`	Guides the blur implementation, experiments, and report.
Starter code set	`BoxBlurSeq.java`, `BoxBlurParallel.java`,
Performance worksheet	A simple table for thread count, runtime, and speedup.

these words interchangeably; correcting that vocabulary early makes later performance discussions much cleaner.

A second misconception is that "more threads always means faster." The runtime table in the slides shows improvement through a small number of threads and then worse performance as thread count rises, illustrating setup cost, scheduling cost, and synchronization cost. This is the right moment to introduce a practical form of Amdahl's Law: the useful work is not the only work the machine performs.

A third misconception is that "One Java statement executes atomically, thus no need to protect the update of every brightness bin for the parallel histogram program as in the *synchronized (lock) hist[v]++;* . We need to explain that a single Java statement is divided into multiple separate instructions with the read and write separated by a gap time.

A fourth misconception is that one-task-per-row is always the best partitioning strategy. The original lab asks students to try multiple rows per task as an experiment. Make that extension visible from the start: row-level tasks are easy to teach, but chunked rows often reduce scheduling overhead and become a natural segue into load balancing and granularity.

Timing Plan

Table 10.4 outlines a recommended timing plan designed to implement this activity across two class sessions. Because course schedules vary, instructors can easily adapt this framework to their unique teaching constraints by truncating or expanding individual segments as needed.

10.5 Course-wide Pre and Post Course Surveys

Development and testing teams administered the following common pre/post survey for CS2. The survey asked students their background knowledge in computing topics, data structures, algorithms, and PDC concepts. Students were asked to rate their prior experience on a 7-point Likert scale:

1. No Experience At All
2. Very Little Experience
3. Slightly Experienced

4. Between Slightly/Moderately Experienced
5. Moderately Experienced
6. Very Experienced
7. Extremely Experienced

The survey consisted of the following 23 items. Adopters can modify the pre/post surveys for background knowledge in computing topics, data structures, and algorithms expected from the students in their courses. Similarly, depending on which PDC topics they intend to introduce, they can choose parallel processing, distributed computing, and event handling.

1. Classes & Objects
2. List data structures
3. Generics (parameterized polymorphism)
4. Stacks
5. Queues
6. Sorting Algorithms
7. Comparators
8. Recursion
9. Maps
10. Sets
11. Trees
12. Binary Search Trees
13. Priority Queues
14. Hashing
15. Graphs
16. Writing Unit Tests & Test Cases
17. Big-O Notation
18. Choosing the correct data structures to solve a problem
19. Designing & Writing Classes
20. Problem Decomposition: Breaking the desired functionality into separate parts (e.g. methods) that can be implemented separately
21. Parallel Processing: Computing where the operations within a program are being executed simultaneously (i.e. running on different cores of the processor at the same time)
22. Distributed Computing: Computing with a program that runs using data from a different computer (i.e. an online service)
23. Event handling: Having a function that responds to an independent action (i.e. a mouse click)

Student attitudes were measured by asking students to agree/disagree with the following five statements.

1. In general, my interest in computing is high
2. My interest in learning about parallel and distributed computing is high
3. In general, I believe learning about parallel and distributed computing is important
4. I believe I have a good understanding of computing
5. I believe I have a good understanding of parallel and distributed computing

The following 7-point Likert scale was used:

1. Strongly Disagree
2. Disagree
3. Somewhat Disagree
4. Neutral
5. Somewhat Agree
6. Agree
7. Strongly Agree

Table 10.4: Timing Plan for a Two-Session Implementation.

Phase	Time	Instructor actions	Student actions	Checkpoint
Session 1: Warm-up	5–10 min	Introduce the goal and run a small grid partitioning.	Predict how one, two, and four workers divide pixels.	Completed partitioning sketch.
Session 1: Concept lecture	20–25 min	Present sequential, parallelism, threads, cores, race conditions, synchronization, speedup.	Multiple choices questions or discussion.	Knowledge of PDC vocabulary.
Session 1: Image-processing demo for edge detection and histogram	20–25 min	Demonstrate kernels, thread counts, output comparison, and nonlinear performance.	performance and correctness.	
Session 1: Code orientation	15–20 min	Walk through file structure, starter code, input/output folders, and timing method.	Compile the sequential file and confirm running environment.	Environment check done.
Session 2: Sequential baseline	10–15 min	Review the 3×3 blur and border handling.	Run `BoxBlurSeq` and save timing.	Sequential runtime and output image.
Session 2: Parallel version	20–25 min	Guide students through thread pool, row tasks, and `CountDownLatch`.	Complete `BoxBlurParallel` and test thread counts.	Parallel output and timing.
Session 2: Experiments	15–20 min	Prompt students to vary thread counts and chunk sizes.	Compute speedup.	Completed performance worksheet.
Session 2: Debrief	10–15 min	Facilitate discussion using the post-lab debrief artifact.	Submit lab reflection.	Lab reflection.

10.6 Conclusion

The activities presented in this chapter demonstrate how PDC concepts can be meaningfully integrated into CS2 by building on students' growing programming skills and understanding of algorithms and data structures. Across the institutional implementations, the approaches connect PDC concepts to established CS2 topics such as recursion, sorting algorithms, arrays, object-oriented programming, event handling, and algorithm analysis. Unplugged activities, visualizations, and programming assignments let students explore task decomposition, concurrency, synchronization, race conditions, load balancing, speedup, and parallel overhead. Activities involving fork-join programming, parallel sorting, event-driven applications, and image processing allow students to move beyond conceptual understanding and gain practical experience implementing and evaluating parallel solutions. A central lesson is that CS2 can incorporate PDC by extending familiar programming problems rather than introducing parallel computing as a separate topic. For instructors interested in adopting or adapting these activities, a good approach is to start with familiar CS2 concepts and gradually introduce parallel implementations, letting students compare sequential and parallel approaches and examine the trade-offs. Unlike CS1, where unplugged activities often provide the primary introduction to PDC, CS2 offers greater opportunities for hands-on programming and performance experimentation. However, differences in course content, programming languages, student preparation, instructional time, and available computing resources influence which activities are most appropriate. The institutional chapters and ac-

companying instructional materials provide examples, starter code, assignments, assessment strategies, and implementation guidance to support adaptation across different CS2 environments. Together, these resources illustrate how PDC can reinforce existing CS2 learning objectives while preparing students for more advanced topics in parallel and distributed computing.

Chapter 11

CS2 Course Package – Tennessee Technological University

Large-Section C++ Infusion with Plugged-In Activities using OpenMP[†]

April R. Crockett[1], Gerald C. Gannod[2]

[1]Tennessee Technological University, acrockett@tntech.edu
[2]Tennessee Technological University, jgannod@tntech.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

11.1 Introduction . . . 325
11.2 How CS2 was changed and PDC topics integrated . . . 325
11.2.1 PDC Topics Integrated . . . 325
11.2.2 Integrated Syllabi (Pre and Post) . . . 325
11.2.3 PDC Topics Integrated . . . 327
11.2.3.1 Module 4 New PDC Content . . . 327
11.2.4 Highlights . . . 327
11.2.5 Challenges . . . 327
11.2.5.1 Assignment that Didn't Work . . . 327
11.2.5.2 Preparing the Teaching Team for New Territory . . . 328
11.3 New Plugged-in Activity . . . 328
11.3.1 Plugged-In Sorting Algorithm Comparison . . . 328
11.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 328
11.3.1.2 Instructional Material (Lectures, Assignments) . . . 332
11.4 Course-wide Pre- and Post-Course Surveys . . . 332
11.4.1 Results of Pre-Course and Post-Course Surveys . . . 332
11.4.1.1 Growth in Core Programming Concepts . . . 332
11.4.1.2 Growth in PDC Concepts . . . 333
11.4.1.3 Summary of Survey Results . . . 334
11.5 Conclusion . . . 335
Appendix . . . 339

Chapter figures

11.1 CS2 Program 2 . . . 328
11.2 CS2 Program 2 Part 1 . . . 329
11.3 CS2 Program 2 Part 2 . . . 330

Chapter tables

11.1 CS2 Course Learning Outcomes & Blooms Levels . . . 322
11.2 CS2 Learning Outcomes & Blooms Levels . . . 323
11.3 CS2 Majors . . . 323
11.4 CS2 Student Gender Identity . . . 324
11.5 CS2 Race/Ethnicity of Students . . . 324
11.6 Classification of CS2 Students . . . 324
11.7 Spring 2026 Course Syllabus Calendar with Highlighted PDC Content . . . 326
11.9 CS2 Program 2 Feedback from Students . . . 331
11.10 CS2 Sample Size for Pre-Course and Post-Course Surveys . . . 332
11.11 CS2 Student Comfort With Core Topics . . . 333
11.12 CS2 Student Comfort With PDC Topics . . . 334

Abstract

This chapter presents a transformation of the second course in the introductory programming sequence (CS2) at Tennessee Technological University in the Department of Computer Science named "Data Structures and Algorithms." This chapter also provides access to course materials so that other institutions may be able to adopt and use the materials. Parallel and Distributed Computing (PDC) concepts were introduced into all sections of CS2 starting in Spring 2025 through lecture discussions and (plugged) programming assignments. The primary learning outcomes are for students to (1) recognize terminology associated with parallel and distributed computing topics, (2) identify algorithmic solutions that are representative of parallelism, and (3) develop programming solutions that are representative of data parallelism.

Descriptor Elements

Programming Language Employed: C++

Relevant core courses: CS2

Pre-requisites: CS1

Textbook: Open Access Resources

Relevant PDC topics and Blooms level Data & Task Parallelism (Blooms levels: Understand, Analyze & Apply)

Learning outcomes: Table 11.1 shows the learning outcomes added to Tennessee Tech's CS2 course with the incorporation of Parallel Computing topics. Table 11.2 shows the activity-level learning outcomes added to Tennessee Tech's CS2 course with the incorporation of Parallel & Distributed Computing topics. Blooms levels are listed for each module and activity (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV) and Creating (CR)).

Table 11.1: CS2 Course Learning Outcomes & Blooms Levels

Learning Outcomes	Blooms Level
Recognize parallel computing terms such as data parallelism, task parallelism, task decomposition, speedup, scalability, and pipelining.	Level 2: Understand
Demonstrate the ability to identify algorithmic solutions that are representative of data parallelism.	Level 4: Analyze
Implement algorithmic solutions that are representative of data parallelism.	Level 3: Apply

Table 11.2: CS2 Learning Outcomes & Blooms Levels

PDC Concept	Module Content & Blooms Level	
	OpenMP Lecture Slides 11.2.3.1	Parallel vs. Sequential Sort Algorithm Comparison 11.3.1
PDC Vocabulary	RE	RE
Data Parallelism	UN	AP,EV

Context for use: Tennessee Technological University (TNTECH) is a mid-sized public research university in Cookeville, Tennessee, United States. With an enrollment of about 10,000 students, TNTECH offers more than 40 undergraduate and 20 graduate degree programs across eight academic colleges and schools. The Department of Computer Science at TNTECH offers Bachelors, Masters, and Ph.D. degrees and consists of approximately 700 total students. At present, we have an average of about 120 students each semester enrolled in our "Data Structures & Algorithms" (CS2) course. The course is a 4 credit-hour lecture-lab, with 3 hours devoted to lecture and 1 hour devoted to lab. The course is usually split into two sections of lecture and four to six sections of lab, where each lab section consists of approximately 25 students each. C++ is the programming language used for instruction, and this course is the second required computer science class in the curriculum of the Computer Science and AI undergraduate degree programs. The CS2 course is taken by students in several additional majors as well, as shown in Table 11.3. Eighty percent of the students in CS2 identify as male and approximately 15% identify as female as shown in Table 11.4 and student race is shown in Table 11.5. Table 11.6 shows the classification of students taking the CS2 course.

Table 11.3: CS2 Majors

Major	**Spr 2025** n=52	**Fall 2025** n=116	**Spr 2026** n=151	**Total** n=319
Business Info. Tech.	0	1	0	1 (0.3%)
Computer Engineering	2	16	19	37 (11.6%)
Computer Science	40	52	70	162 (50.8%)
Electrical Engineering	8	41	53	102 (32.0%)
Environmental Engineering	1	0	0	1 (0.3%)
Finance	0	0	1	1 (0.3%)
Mathematics	2	2	5	9 (2.8%)
Mechanical Engineering	0	1	2	3 (0.9%)
Music	0	0	1	1 (0.3%)
Nuclear Engineering	0	1	1	2 (0.6%)
Physics	1	3	8	12 (3.8%)
Spanish	1	0	0	1 (0.3%)

Table 11.4: CS2 Student Gender Identity

Gender	**Spr 2025** n=52	**Fall 2025** N=116	**Spr 2026** N=151	**Total** N=319
Male	35	91	121	247 (77.4%)
Female	13	14	19	46 (14.4%)
Other	1	2	1	4 (1.3%)
No Response	3	9	10	22 (6.9%)

Table 11.5: CS2 Race/Ethnicity of Students

Race/Ethnicity	**Spr 2025**	**Fall 2025**	**Spr 2026**	**Total**
White	31	78	110	219 (68.7%)
Asian/ Pacific Islander	10	6	9	25 (7.8%)
Black/ African American	6	3	12	21 (6.6%)
Prefer Not to Respond	3	11	4	18 (5.6%)
Hispanic	1	7	9	17 (5.3%)
2 or More Ethnicities	0	9	6	15 (4.7%)
American Indian/ Alaska Native	0	1	1	2 (0.6%)
Middle Eastern/ North African	0	1	0	1 (0.3%)

Table 11.6: Classification of CS2 Students

Classification	**Spr 2025** N=51	**Fall 2025** N=115	**Spr 2026** N=150	**Total** N=316
First year (freshman)	24	3	55	82 (25.9%)
Sophomore	17	81	62	160 (50.6%)
Junior	8	27	29	64 (20.3%)
Senior	2	3	4	9 (2.8%)
Graduate student	0	1	0	1 (0.3%)

11.1 Introduction

Modern computing increasingly relies on parallelism, distributed systems, and data-intensive workflows. National curriculum recommendations such as the *ACM/IEEE Computer Science Curricula* and the *IEEE TCPP Parallel and Distributed Computing (PDC)* guidelines, have highlighted the growing relevance of these topics within undergraduate computing education [35, 51]. At *Tennessee Technological University*, we have infused parallel and distributed computing (PDC) concepts into our undergraduate computing curricula staring with our CS1 course and continuing on in our CS2 course.

Section 11.2 below describes in detail what was changed in our CS2 course and what specifically was added. We modified a few of our lectures and modified some assignments to incorporate Parallel Computing topics. Section 11.2.4 highlights the changes and section 11.2.5 discusses challenges faced in updating the curriculum. The details on all new activities is in section 11.3.

11.2 How CS2 was changed and PDC topics integrated

To add PDC topics in CS2 at TNTECH, we modified an existing assignment so that traditional CS2 objectives were retained while parallel computing concepts were embedded into the study of sorting algorithms. Specifically, a programming assignment formerly centered on students writing and comparing time efficiencies of sort algorithms was revised so that students used OpenMP to additionally write and compare parallel sort algorithms.

11.2.1 PDC Topics Integrated

In our CS2 course, parallel programming and OpenMP was the lecture topic during week 4 of lecture and practiced via a programming assignment during weeks 5 through 7.

11.2.2 Integrated Syllabi (Pre and Post)

A syllabus and master schedule for each semester are available in the GitHub repository here. CS2 is named CSC 1310 at TNTECH. The schedule and syllabus for Spring 2025 is before PDC topics were introduced and all other semesters are after. The course calendar from the syllabus for CS2 at TNTECH as shown in Table 11.7 spans 16 weeks across 9 to 10 content modules, covering classes, recursion, templates, genAI/prompt engineering, algorithm analysis, search and sort algorithms, lists, stacks, queues, hash tables, binary search trees, graphs, balanced trees, and heaps. The 16th week is final exams week. The PDC interventions, highlighted in the course calendar, are currently in Module 4, with plans to extend further into other modules.

Table 11.7: Spring 2026 Course Syllabus Calendar with Highlighted PDC Content

Module	Week	Assignments	PDC Content Added	Quiz
Mod 1: OOP, Classes, Objects, Namespaces, & Makefiles	Weeks 1–3	Syllabus, Informed Consent Form, Pre-Course Experience Survey, Program 1 (OOP), Lab 1 Assignment, Lab 2 Assignment	N/A	Mod 1 Quiz
Mod 2: Recursion & Templates	Weeks 3–4	Lab 3 Assignment	N/A	Mod 2 Quiz
Mod 3: GenAI & Prompt Engineering	Week 4	GenAI and Prompt Engineering instructional activities	N/A	N/A
Mod 4: Algorithm Analysis, Searching/Sorting, Parallel Programming & OpenMP	Weeks 5–6	Module 4 Practice, Lab 4 on Sorting Algorithms, Lab 5 on Program 2 Work Session, Program 2 Sorting Algorithms & Parallel Programming with OpenMP	Parallel programming and OpenMP instruction, OpenMP examples and activities, and Program 2 on Sorting Algorithms & Parallel Programming with OpenMP (Mod 4 PDC Content)	Mod 4 Quiz
Mod 5: Lists, Stacks, & Queues	Weeks 7–8	Module 5 Practice, Lab 6 Assignment, Lab 7 Assignment, Program 3	N/A	Mod 5 Quiz
Mod 6: Hash Tables	Weeks 8–10	Module 6 Practice, Lab 8 Assignment	N/A	Mod 6 Quiz
Mod 7: Trees & Binary Search Trees	Weeks 10–11	Trees and Binary Search Tree activities, Lab 9 Assignment, Program 4	N/A	Mod 7 Quiz
Mod 8: Graphs	Weeks 11–14	Module 8 Practice, Lab 10 Assignment, Program 4	N/A	Mod 8 Quiz
Mod 9: Balanced Trees & Heaps	Weeks 14–15	Module 9 Practice, Post-Course CSC Programming Survey, Post-Course Experience Survey	N/A	N/A
Course Wrap-Up & GenAI Workshop	Week 16	Coding with GenAI Lab/Workshop, IDEA Evaluation, Final Exam Review, Post-Course Surveys	N/A	N/A

11.2.3 PDC Topics Integrated

11.2.3.1 Module 4 New PDC Content

In our CS2 course, students are traditionally introduced to algorithm efficiency, including time complexity and Big-O notation. They also study common searching and sorting algorithms and learn to compare and contrast their time and space complexity. The PDC content integrated into this module builds on these concepts by demonstrating how divide-and-conquer algorithms, such as merge sort and quicksort, are well suited for parallel execution. Students first examine how quicksort can be decomposed into independent subproblems that may be executed concurrently. They are then introduced to OpenMP and learn how to use its parallel programming constructs to implement a parallel version of quicksort. This integration allows students to connect their existing understanding of algorithm design and complexity with the potential performance benefits and practical considerations of parallel execution. Here is a list of materials that were used in Module 4:

- PRESENTATION SLIDES: Search Analysis Sort.pptx
- Module 4: Example Programs.zip
- PRESENTATION SLIDES: CS2 OpenMP Introduction.pptx
- OpenMP Examples

We then paired the lecture with an assigned Plugged Activity (Programming Assignment) which is detailed in 11.3.1. Here is a list of materials that were used as part of this assignment.

- CSC 1310 Program 2 Assignment Spring 2026.pdf
- Rubric Part 1
- Rubric Part 2
- Given Files: CSC 1310 Program 2 Spring 2026.zip

11.2.4 Highlights

Much of the existing CS2 course remained unchanged following the integration of PDC content. We modified an existing assignment so that traditional CS2 objectives were retained while parallel computing concepts were embedded into the study of sorting algorithms. Specifically, a programming assignment formerly centered on students writing and comparing time efficiencies of sort algorithms was revised so that students used OpenMP to additionally write and compare parallel sort algorithms.

11.2.5 Challenges

11.2.5.1 Assignment that Didn't Work

In Fall 2025 we tried incorporating accessing data from a remote service via a lab assignment in the course. You can view the assignment here. The assignment ended up being too open-ended. However, with some modifications, we plan to add this assignment back to the course. The problem was that the assignment was too open-ended in allowing students to find a *JSON* data repository to use but many students ran into the problem of finding repositories that required a key or repositories with data issues.

11.2.5.2 Preparing the Teaching Team for New Territory

Another challenge was that the new material was not only new to the students in the course, but also to the team of teaching assistants. This meant that we would first need to find time to teach the concepts and assignments to the teaching assistants before teaching the students. Finding the time to do this was definitely challenging. There were also some teaching assistants that didn't understand why these concepts were being taught in CS2 and thought it was too much to ask CS2 students to learn about and practice these concepts. We originally didn't expect this confusion and didn't explain the reasoning. However, in the 2nd semester of implementing the PDC concepts we knew that explaining the reasoning must be part of the TA training.

11.3 New Plugged-in Activity

11.3.1 Plugged-In Sorting Algorithm Comparison

11.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

The **prerequisites** for this programming assignment include understanding algorithm analysis and sort algorithms. At Tennessee Tech, we cover these topics in week 4 (out of 16) and then students are assigned this 3-week long programming assignment during week 5 of the semester.

Figure 11.1: CS2 Program 2

PROGRAM 2

CSC 1310, SPRING 2026

SORTING ALGORITHM COMPARISON

OBJECTIVE OF THIS ASSIGNMENT



The primary goal & challenge of this assignment is not just learning to write code but in learning about sort algorithms, comparing algorithm efficiencies, and learning to apply parallel programming using OpenMP.

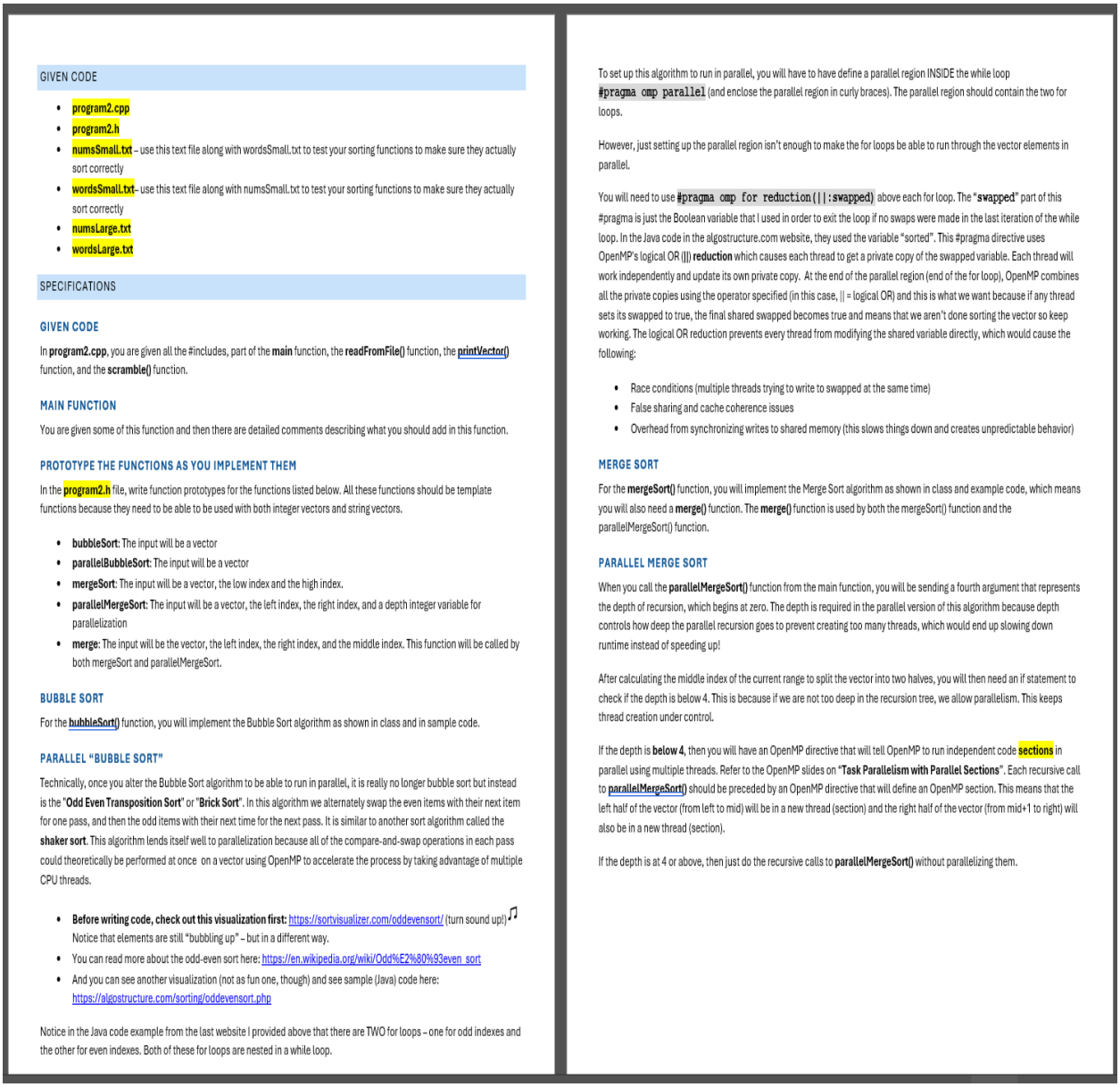
Modern software is built to run fast and efficiently on multi-core systems, and that means parallel programming is ***no longer optional—it's essential****. Learning it early* ***gives you a serious edge****, making you more competitive, job-ready, and prepared to solve real-world problems at scale.*

Image created with ChatGPT 4.0 https://chatgpt.com/s/m_68682730c4b48191978c1a469a4cb100

TWO DUE DATES!

- Part 1: The working program code is due on **3/3/2026**
- Part 2: The testing documentation document is due on **3/5/2026**

Figure 11.2: CS2 Program 2 Part 1



GIVEN CODE

- program2.cpp
- program2.h
- numsSmall.txt – use this text file along with wordsSmall.txt to test your sorting functions to make sure they actually sort correctly
- wordsSmall.txt– use this text file along with numsSmall.txt to test your sorting functions to make sure they actually sort correctly
- numsLarge.txt
- wordsLarge.txt

SPECIFICATIONS

GIVEN CODE

In **program2.cpp**, you are given all the #includes, part of the **main** function, the **readFromFile()** function, the **printVector()** function, and the **scramble()** function.

MAIN FUNCTION

You are given some of this function and then there are detailed comments describing what you should add in this function.

PROTOTYPE THE FUNCTIONS AS YOU IMPLEMENT THEM

In the **program2.h** file, write function prototypes for the functions listed below. All these functions should be template functions because they need to be able to be used with both integer vectors and string vectors.

- **bubbleSort**: The input will be a vector
- **parallelBubbleSort**: The input will be a vector
- **mergeSort**: The input will be a vector, the low index and the high index.
- **parallelMergeSort**: The input will be a vector, the left index, the right index, and a depth integer variable for parallelization
- **merge**: The input will be the vector, the left index, the right index, and the middle index. This function will be called by both mergeSort and parallelMergeSort.

BUBBLE SORT

For the **bubbleSort()** function, you will implement the Bubble Sort algorithm as shown in class and in sample code.

PARALLEL "BUBBLE SORT"

Technically, once you alter the Bubble Sort algorithm to be able to run in parallel, it is really no longer bubble sort but instead is the "**Odd Even Transposition Sort**" or "**Brick Sort**". In this algorithm we alternately swap the even items with their next item for one pass, and then the odd items with their next time for the next pass. It is similar to another sort algorithm called the **shaker sort**. This algorithm lends itself well to parallelization because all of the compare-and-swap operations in each pass could theoretically be performed at once on a vector using OpenMP to accelerate the process by taking advantage of multiple CPU threads.

- **Before writing code, check out this visualization first:** https://sortvisualizer.com/oddevensort/ (turn sound up!) ♫ Notice that elements are still "bubbling up" – but in a different way.
- You can read more about the odd-even sort here: https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort
- And you can see another visualization (not as fun one, though) and see sample (Java) code here: https://algostructure.com/sorting/oddevensort.php

Notice in the Java code example from the last website I provided above that there are TWO for loops – one for odd indexes and the other for even indexes. Both of these for loops are nested in a while loop.

To set up this algorithm to run in parallel, you will have to have define a parallel region INSIDE the while loop `#pragma omp parallel` (and enclose the parallel region in curly braces). The parallel region should contain the two for loops.

However, just setting up the parallel region isn't enough to make the for loops be able to run through the vector elements in parallel.

You will need to use `#pragma omp for reduction(||:swapped)` above each for loop. The "**swapped**" part of this #pragma is just the Boolean variable that I used in order to exit the loop if no swaps were made in the last iteration of the while loop. In the Java code in the algostructure.com website, they used the variable "sorted". This #pragma directive uses OpenMP's logical OR (||) **reduction** which causes each thread to get a private copy of the swapped variable. Each thread will work independently and update its own private copy. At the end of the parallel region (end of the for loop), OpenMP combines all the private copies using the operator specified (in this case, || = logical OR) and this is what we want because if any thread sets its swapped to true, the final shared swapped becomes true and means that we aren't done sorting the vector so keep working. The logical OR reduction prevents every thread from modifying the shared variable directly, which would cause the following:

- Race conditions (multiple threads trying to write to swapped at the same time)
- False sharing and cache coherence issues
- Overhead from synchronizing writes to shared memory (this slows things down and creates unpredictable behavior)

MERGE SORT

For the **mergeSort()** function, you will implement the Merge Sort algorithm as shown in class and example code, which means you will also need a **merge()** function. The **merge()** function is used by both the mergeSort() function and the parallelMergeSort() function.

PARALLEL MERGE SORT

When you call the **parallelMergeSort()** function from the main function, you will be sending a fourth argument that represents the depth of recursion, which begins at zero. The depth is required in the parallel version of this algorithm because depth controls how deep the parallel recursion goes to prevent creating too many threads, which would end up slowing down runtime instead of speeding up!

After calculating the middle index of the current range to split the vector into two halves, you will then need an if statement to check if the depth is below 4. This is because if we are not too deep in the recursion tree, we allow parallelism. This keeps thread creation under control.

If the depth is **below 4**, then you will have an OpenMP directive that will tell OpenMP to run independent code **sections** in parallel using multiple threads. Refer to the OpenMP slides on "**Task Parallelism with Parallel Sections**". Each recursive call to **parallelMergeSort()** should be preceded by an OpenMP directive that will define an OpenMP section. This means that the left half of the vector (from left to mid) will be in a new thread (section) and the right half of the vector (from mid+1 to right) will also be in a new thread (section).

If the depth is at 4 or above, then just do the recursive calls to **parallelMergeSort()** without parallelizing them.

Figure 11.3: CS2 Program 2 Part 2

PART 2: DOCUMENTATION

In a Word document named **program2_documentation.docx**, you must document going through the following steps to test your code **after you know it is working properly**. Read incremental programming & testing above!

1. First, make sure to **disable anti-virus** on your computer and **stop/close all unnecessary programs** from running (close out Teams, Outlook, Discord, etc.). You may keep Microsoft Word or Google Docs open to document the steps below.
2. **Run Sample Output 1** (shown in Sample Output section in this document).
3. Take a **screenshot** of your sample output 1 results and paste into the document.
4. **Analyze your results** and write answers to the questions below in the document.
    a. Why do you think you got the results that you got?
    b. What algorithm was the fastest out of the algorithms you implemented?
    c. What algorithm was the slowest?
    d. Was the parallel bubble sort algorithm faster than the serial bubble sort algorithm?
    e. How does your results compare to the sample output provided?
5. **Run Sample Output 2** (shown in next section), including the input validation.
6. Take a **screenshot** of your sample output 2 results and paste into the document.
7. **Analyze your results**.
    a. Why do you think you got the results that you got?
    b. How does it compare to the sample output provided?
8. **Run Sample Output 3** (shown in next section).
9. Take a **screenshot** of your sample output 3 results and paste into the document.
10. **Analyze your results**.
    a. Why do you think you got the results that you got?
    b. How does it compare to the sample output provided?
11. **Try several different numbers of threads** and figure out what number of threads gave you the quickest runtime for the parallel algorithms. Provide a **screenshot** of the quickest runtime of parallel bubble sort.
12. You may have noticed that when you increase the number of threads that parallel bubble sort slows down significantly, but parallel merge sort does not. **Why is that?** (Note: you may use ChatGPT or similar GenAI to help you with this answer if needed).

Students filled out a program report after every programming assignment. Students rate their level of agreement with the three statements shown in Table 11.9. Overall, students responded positively to the assignment, with the strongest result being that students felt they learned from the assignment. A large majority of students agreed at least somewhat that they learned from it, with 87.2% selecting Somewhat Agree, Agree, or Strongly Agree, and only 1.4% expressing disagreement. Students were also generally confident in their performance, with 81.8% agreeing that they thought they made a good grade.

Table 11.9: CS2 Program 2 Feedback from Students

Statement	**Agree** (Strongly Agree, Agree, Somewhat Agree)	**Neutral**	**Disagree** (Somewhat Disagree, Disagree, Strongly Disagree
I enjoyed this assignment	110 (74.3%)	27	11 (7.4%)
I learned from this assignment	129 (87.2%)	17	2 (1.4%)
I think I made a good grade on this assignment	121 (81.8%)	20	7 (4.7%)

Several open-ended questions were also asked. One of the questions asked was "What is something you learned from the assignment?" Students' responses suggest that the assignment primarily helped them understand OpenMP, parallel programming, sorting algorithms, and performance tradeoffs. The strongest theme was that many students learned how to apply OpenMP syntax and parallelize code, especially through `#pragma`, parallel sections, task parallelism, thread usage, and compiler flags such as `-fopenmp`. A second major theme was that students deepened their understanding of sorting algorithms, particularly bubble sort, merge sort, quicksort, and how different algorithms behave with large datasets. Several students said that implementing the algorithms themselves helped them better understand how the algorithms work, how loops and variables function within them, and how sorting methods differ in efficiency. Another important finding is that many students learned that parallelization does not automatically improve performance. Several responses specifically noted that parallel code can be slower than serial code, especially for small datasets or when thread creation, synchronization, or coordination overhead outweighs the benefit. Students also observed that more threads do not always mean faster runtime, and that there may be an optimal number of threads depending on the algorithm and dataset size. Students also reported learning how to measure and compare runtime, including use of the *chrono* library, best-case and worst-case runtime differences, and how performance can vary across devices. A few students connected the assignment to broader ideas about algorithm efficiency, data type effects, and the strengths and weaknesses of different sorting approaches. There were also several responses indicating that the assignment was challenging. Some students described OpenMP or parallel programming as confusing, strange, difficult, or frustrating, while still acknowledging that they were improving. Overall, the responses indicate that the assignment was successful in helping many students connect sorting algorithm implementation with parallel programming concepts and empirical performance testing. The most meaningful learning outcome appears to be that students moved beyond seeing parallelism as simply "faster" and began recognizing that performance depends on the algorithm, dataset size, number of threads, overhead, and hardware.

11.3.1.2 Instructional Material (Lectures, Assignments)

- Lecture Materials on Algorithm Analysis, Sort Algorithms, and Parallel Programming: url
- Assignment Document and Files Given to Students: url

11.4 Course-wide Pre- and Post-Course Surveys

To determine if our new and modified modules and assignments were effective at teaching basic PDC concepts while not reducing understanding of core, fundamental CS2 topics, we administered pre- and post-course surveys §10.5. To download the actual surveys performed in CS2 at TNTECH, go to url.

Table 11.10 shows how many students were enrolled in the CS2 course from Spring 2025 to Spring 2026. The table also shows how many students filled out the Pre-Course survey and Post-Course survey each semester who also consented to share their data in the study. These students' results are those reported in Section 11.4.1.

Table 11.10: CS2 Sample Size for Pre-Course and Post-Course Surveys

Population	Spr 2025	Fall 2025	Spr 2026	Total
No. Students Enrolled in CS2 Course	60	132	168	**360**
No. Consented Students Pre-Course Survey	52	116	151	**319 (88.6%)**
No. Consented Students Post-Course Survey	51	115	150	**316 (87.8%)**

11.4.1 Results of Pre-Course and Post-Course Surveys

All institutions used a common Pre-Course and Post-Course survey §10.5 which asked students to rate their comfort level with 19 computing and 4 PDC-related concepts.

11.4.1.1 Growth in Core Programming Concepts

As shown in Table 11.11, across the Spring 2025 through Spring 2026 intervention period, students reported large pre-to-post gains in core data structures and algorithm concepts. We had 285 students complete both the pre-course and post-course surveys. In this table, pre-mean is the mean pre-course response score for the students included in that item's analysis. Post-mean is the mean post-course response score for the same students included in the pre-course mean. Diff is the mean paired change from pre-course to post-course, calculated as Post-Mean - Pre-Mean. Positive values indicate an increase from pre to post. *Wilcoxon* is the *Wilcoxon* signed-rank test statistic for the paired pre/post responses. This nonparametric test evaluates whether the median paired change differs from zero. The p-value from the Wilcoxon signed-rank test is also displayed in the table. Smaller values indicate stronger evidence that the pre/post change was not due to random variation. The "sig?" column indicates whether the *Wilcoxon* test result was statistically significant (yes or no) using

alpha level .05. A value of yes means p is less than 0.05 and no means p is greater than or equal to .05.

Table 11.11: CS2 Student Comfort With Core Topics

Topic	Pre-mean	Post-Mean	Diff	Wilcoxon	p-value	sig?
Classes & Objects	3.98	5.63	1.65	287.00	1.65E-38	Yes
Searching algorithms	2.39	5.11	2.72	346.00	5.12E-44	Yes
Sorting algorithms	2.39	5.11	2.72	427.00	8.64E-44	Yes
Recursion	2.24	4.89	2.65	617.00	8.98E-42	Yes
Template Classes/ Generics	2.14	5.11	2.97	315.00	4.37E-45	Yes
Lists/ Linked Lists	2.44	5.46	3.02	427.00	4.95E-44	Yes
Stacks	1.76	5.31	3.55	99.00	1.83E-47	Yes
Queues	1.64	5.26	3.62	118.50	2.31E-47	Yes
Priority Queues	1.53	4.44	2.91	268.00	3.99E-45	Yes
Hashing/ Hash Tables	1.60	4.86	3.26	174.00	6.04E-47	Yes
Trees	1.64	5.08	3.45	114.00	5.25E-47	Yes
Binary Search Trees	1.59	5.09	3.50	168.00	7.74E-47	Yes
Balanced Trees/ AVL Trees	1.45	4.63	3.18	54.00	2.15E-46	Yes
Heaps	1.56	4.28	2.72	42.00	4.94E-45	Yes
Treap	1.37	3.53	2.17	345.00	1.84E-39	Yes
Graphs	1.76	4.85	3.09	104.50	7.12E-46	Yes
B-Trees	1.38	3.75	2.37	51.00	1.81E-41	Yes
Sets	1.68	3.65	1.97	977.00	9.37E-35	Yes
Associative Array, Map, or Dictionary	1.84	3.93	2.08	905.00	5.78E-37	Yes
Writing test cases	2.07	4.61	2.54	618.00	9.03E-42	Yes
Big-O notation	1.69	4.87	3.18	282.50	2.67E-46	Yes
Choosing the correct data structure	2.38	5.27	2.89	323.00	3.50E-44	Yes
Writing Classes	3.07	5.74	2.68	75.00	8.61E-45	Yes
Problem decomposition	3.64	5.46	1.82	1280.00	5.58E-35	Yes

11.4.1.2 Growth in PDC Concepts

As shown in Table 11.12, students reported significant gains in parallel processing (largest gain), distributed computing, and event handling (smallest gain). In the table, Pre-Mean is the mean pre-course response score for the students included in that item's analysis. Post-Mean is the mean post-course response score for the same students included in the pre-course mean. Diff is the mean paired change from pre-course to post-course, calculated as Post Mean - Pre-Mean. Positive values indicate an increase from pre to post. *Wilcoxon* is the *Wilcoxon* signed-rank test statistic for the paired pre/post responses. This nonparametric test evaluates whether the median paired change differs from zero. The p-value from the *Wilcoxon* signed-rank test is also displayed in the table. Smaller values indicate stronger evidence that the pre/post change was not due to random variation. The "sig?" column indicates whether the *Wilcoxon* test result was statistically significant (yes or no) using alpha level .05. A value of yes means p is less than 0.05 and no means p is greater than or equal to .05.

Table 11.12: CS2 Student Comfort With PDC Topics

Topic	pre-mean	Post Mean	Diff	Wilcoxon	p-value	sig?
Parallel Processing: Computing where the operations within a program are being executed simultaneously	2.68	4.40	1.73	1102.00	1.94E-37	Yes
Distributed Computing: Computing with a program that runs on multiple computers or that uses data from a different computer	2.28	3.98	1.70	1492.50	8.26E-34	Yes
Event Handling: Having a function that responds to an independent action	2.74	4.21	1.47	2905.00	1.26E-25	Yes

11.4.1.3 Summary of Survey Results

The course-wide pre- and post-course survey results indicate that students reported statistically significant increases in their comfort with both foundational CS2 topics and PDC concepts during the Spring 2025 through Spring 2026 intervention semesters. Of the 360 students enrolled across these semesters, 285 completed matched pre- and post-course surveys and were included in the paired analysis. As shown in Table 11.10, students reported significant gains across all 24 traditional CS2 topics assessed, including sorting algorithms, recursion, linked lists, stacks, queues, trees, graphs, and algorithm analysis. These findings are consistent with continued development of traditional CS2 knowledge during the PDC-infused course, although the surveys do not establish whether the same gains would have occurred without the intervention.

Students also reported statistically significant growth in all three PDC-related concepts presented in Table 11.12. Parallel processing had the largest mean paired increase (1.73 points), followed closely by distributed computing (1.70 points) and event handling (1.47 points). The increase in parallel processing is particularly relevant because the primary PDC intervention explicitly incorporated OpenMP and parallel sorting into an existing algorithm-analysis assignment. The gains in distributed computing and event handling should be interpreted more cautiously because these topics received less sustained instructional attention in the reported CS2 activities. All three survey findings reflect changes in students' perceived comfort rather than direct measurements of independent PDC proficiency.

Student feedback on the modified sorting assignment provides complementary, activity-level evidence. Of the 148 students represented in the assignment feedback, 87.2% agreed at least somewhat that they learned from the assignment, 74.3% reported enjoying it, and 81.8% believed they had earned a good grade. Open-ended responses described learning to use OpenMP directives, compare sequential and parallel sorting implementations, measure execution time, and recognize that parallel execution is not always faster. Students also identified difficulties with OpenMP syntax, development tools, and parallel programming concepts. Together, these responses suggest that students engaged with the intended performance-analysis concepts while identifying areas where clearer scaffolding would improve implementation.

Overall, the results support the feasibility of introducing practical PDC experiences within a traditional, C++-based CS2 course without replacing its core data structures and

algorithms content. They do not, however, establish a causal effect of PDC infusion on CS2 achievement or demonstrate mastery of parallel programming. The most defensible conclusion is that students reported increased familiarity with both traditional and PDC concepts and that the parallel sorting assignment gave many students an opportunity to connect algorithm analysis with empirical performance behavior.

11.5 Conclusion

Parallel and distributed computing (PDC) concepts are traditionally introduced in upper-level computer science courses and electives. However, the TNTECH CS2 exemplar demonstrates a practical approach to introducing parallel programming while students continue studying foundational data structures and algorithms. By incorporating OpenMP into an existing module on sorting algorithms and algorithm analysis, students explored parallel execution, task decomposition, and performance tradeoffs without a substantial restructuring of the course. The intervention connected theoretical discussions of algorithm efficiency to measured execution time and the effects of dataset size, thread count, and parallel overhead.

The course-wide surveys showed statistically significant pre-to-post increases in students' self-reported comfort across all assessed traditional CS2 topics and in parallel processing, distributed computing, and event handling. Student feedback on the modified sorting assignment was generally positive, and reflections suggested that many students began recognizing that parallelization does not automatically produce faster programs. These findings provide encouraging evidence of feasibility and perceived learning, but they should not be interpreted as proof of PDC mastery or as establishing that the intervention caused the observed gains.

- **Lessons Learned**
  - One important lesson was that PDC concepts could be incorporated most naturally when they extended topics already taught in CS2. Sorting algorithms and algorithm analysis provided an appropriate setting for introducing OpenMP because students were already implementing algorithms and comparing their performance. Modifying an existing assignment preserved the established learning objectives while introducing parallel execution as an additional dimension of algorithm analysis.
  - A second lesson was the value of comparing sequential and parallel implementations empirically. Student reflections suggested that many learned that parallel programs are not necessarily faster, especially for smaller datasets or when thread-management and coordination costs outweigh potential speedup. These observations made performance tradeoffs more concrete and helped connect classroom discussions of efficiency to measured behavior.
  - The modified sorting assignment also showed that programming infrastructure can complicate an otherwise focused activity. Students had to understand OpenMP directives, thread behavior, and compiler options while completing the sorting and benchmarking tasks. Some students reported that the new syntax and concepts were confusing or frustrating. Providing a working environment and short introductory examples can help prevent tooling difficulties from obscuring the

intended learning outcomes.
  - An attempted remote-data assignment in Fall 2025 highlighted the risks of making an activity too open-ended. Students were allowed to select a JSON repository, but some encountered authentication requirements or inconsistent data that prevented them from completing the intended work. The experience indicated that remote-data activities need carefully selected sources and clearer constraints, particularly when external service configuration is not itself a learning objective.
  - Finally, preparation of the teaching assistants was essential. Because the PDC material was new to some members of the instructional team, they needed time to learn both the technical material and the rationale for including it in CS2. Initial uncertainty about the appropriateness of parallel programming for introductory students led the team to emphasize pedagogical justification during subsequent teaching-assistant training.

- **Teaching Tips for Instructors**
  - Introduce parallel programming as an extension of familiar CS2 topics rather than as an isolated advanced unit. Students should first understand the sequential algorithm and then identify operations or subproblems that may be executed concurrently. Sorting provides an accessible connection among recursion, algorithm analysis, and parallel execution.
  - Provide small OpenMP examples before assigning a larger program. Demonstrate the purpose of the relevant directives, thread behavior, and required compiler options. A short compilation and execution exercise can help distinguish environment problems from misunderstandings of parallel programming concepts.
  - Require students to compare sequential and parallel implementations using a range of input sizes and, where appropriate, thread counts. Ask them to explain results in which additional threads provide little benefit or lead to slower execution. Assessment should reward accurate measurements and sound interpretations rather than expecting parallel code to outperform sequential code in every test.
  - Use consistent benchmarking practices. Students should compare algorithms on equivalent inputs, repeat timing measurements when feasible, and recognize the effects of hardware, system load, and parallel overhead. These practices reinforce experimental reasoning alongside algorithm analysis.
  - Include short written reflection questions in the programming assignment. At TNTECH, student responses provided useful insight into what students believed they learned about OpenMP, sorting algorithms, and performance, as well as where they encountered difficulty. Reflections can expose misunderstandings that are not apparent from working source code alone.
  - When using remote data, give students a verified API or repository with documented response formats and manageable authentication requirements. Starter code may be appropriate for networking or parsing operations that are incidental to the learning goal. Greater choice can be offered once students have completed a structured example successfully.

  - Prepare teaching assistants in advance by having them complete the activities, test compiler configurations, and discuss likely student questions. Training should explain both how an activity works and why it reinforces existing CS2 outcomes, with a short retrospective after the activity to identify needed revisions.

- **Inferences and Limitations**
  - The combined findings suggest that a large, C++-based CS2 course can incorporate parallel programming experiences while students continue reporting substantial growth in traditional data structures and algorithms topics. They provide evidence of feasibility and perceived learning, not proof that PDC infusion independently improved mastery of either traditional or PDC concepts.
  - A primary limitation is that the course-wide surveys measured self-reported comfort rather than directly testing students' ability to design, implement, or explain parallel and distributed programs. Programming artifacts and assignment reflections provide complementary evidence of engagement, but they are not substitutes for a standardized objective assessment of PDC understanding.
  - The reported pre-to-post comparisons also do not isolate the effects of the intervention from general learning over the semester. Students received instruction in many CS2 topics, and the chapter does not report a randomized comparison of PDC-infused and non-PDC sections for the same outcomes. The statistically significant changes therefore should not be interpreted as causal estimates of the intervention's effectiveness.
  - The matched survey analysis represented 285 of the 360 enrolled students across three semesters. Students who did not provide matched, consented responses may have differed from those included in the analysis. Results may also vary in courses with different programming languages, section sizes, laboratory arrangements, teaching-assistant support, or development environments.
  - Although gains were statistically significant for parallel processing, distributed computing, and event handling, the principal documented CS2 activity focused on parallel sorting with OpenMP. Accordingly, the results provide stronger instructional alignment for parallel processing than for the other two PDC topics. Similar increases in self-reported comfort should not be taken to imply equivalent depth of instruction or competence.

- **Future Work**
  - Future work at TNTECH will consider how PDC experiences can be reinforced through additional CS2 topics without weakening core learning objectives.
  - The remote-data activity attempted in Fall 2025 should be revised with verified sources, clearer expectations, and more structured support for JSON access. A narrower initial activity would let students focus on remote data retrieval and processing rather than troubleshooting unrelated API and data-format problems.
  - Further curricular development may explore opportunities to connect CS2 data structures and algorithms to Big Data and artificial intelligence (AI). Activities built around larger datasets, searching, sorting, hashing, trees, and graphs could help students examine how data volume, representation, and algorithm choice

affect computational resources and performance. These are proposed directions rather than outcomes evaluated in the present study.

– Future evaluations should supplement self-reported comfort with direct assessments of PDC knowledge and transfer. Students might be asked to identify independent operations, explain possible race conditions, interpret timing data, or justify when a sequential approach is preferable. Comparisons across course offerings and analyses of prior programming experience could help clarify for whom particular activities are most effective.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter11-TTU

Detailed Pre and Post Syllabus

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter11-TTU/Detailed_Pre_and_Post_Syllabus

Organization of Material

The supporting materials for the TNTECH CS2 course package are available in the TNTECH CS2 GitHub repository. The repository is organized into the following categories:

- **Detailed Post-Intervention Syllabus and Course Schedule**
  The `Detailed_Pre_and_Post_Syllabus` directory contains the detailed Spring 2026 course documents:
  - CSC 1310-001 Spring 2026 Syllabus
  - CSC 1310 Spring 2026 Master Schedule

  These files document the organization of CSC 1310, Data Structures and Algorithms, after the integration of parallel-computing topics and OpenMP. The placement of the PDC material within the course is described in Section 11.2.2, with the new Module 4 content described in Section 11.2.3.1.
- **Plugged Activities**
  The `Plugged_Activities` directory contains the programming and laboratory activities associated with the PDC content discussed in this chapter. It is organized into the following activity directories:
  - **Sorting Algorithms with OpenMP**
    This directory contains the primary PDC-infused programming assignment described in Section 11.3.1. Students build on their study of algorithm analysis and sorting by implementing and comparing sequential and parallel sorting algorithms using OpenMP.
    The directory contains:
    * CSC 1310 Program 2 Assignment Spring 2026 (DOCX and PDF)
    * Program 2 Part 1 grading rubric
    * Program 2 Part 2 grading rubric
    * `Given Files CSC 1310 Program 2 Spring 2026`
    * `Module 4 Analysis & Algorithms & Parallel Computing`

    The `Given Files CSC 1310 Program 2 Spring 2026` directory contains the starter files and datasets supplied to students:

    * program2.cpp
    * program2.h
    * numsSmall.txt
    * numsLarge.txt
    * wordsSmall.txt
    * wordsLarge.txt

  The Module 4 Analysis & Algorithms & Parallel Computing directory contains the lecture and example materials used to prepare students for the assignment:

    * [PRESENTATION SLIDES] Search Analysis Sort.pptx
    * [PRESENTATION SLIDES] CS2 OpenMP Introduction.pptx
    * Module 4 Example Programs
    * OpenMP Examples
    * Supplemental Module 4 image and animation files

  These instructional materials correspond to the content described in Section 11.2.3.1. Student feedback about the programming assignment is summarized in Table 11.9.

- **Lab 5: GitHub Copilot & OOP Design with Binary Search**
  This directory contains the earlier remote-data laboratory activity discussed in Section 11.2.5. The repository currently includes CSC 1310 Fall 2025 LAB 3 Assignment Directions.docx. As discussed in the chapter, this activity was found to be too open-ended in its original form and is retained as an example for possible revision and future use.

- **Evaluation Data and Survey Instruments**
  The Evaluation_Data_and_Analysis-Survey_Instruments directory contains the survey and reporting instruments used during the project. These include informed consent forms, pre-course and post-course experience surveys, lab reports, programming assignment reports, and pre-activity quizzes.
  Materials in the repository include instruments from the following semesters:

  - **Fall 2024**
    * CS1 & CS2 Fall 2024 Informed Consent Form
    * CS1 & CS2 Fall 2024 Lab Report
    * CS2 Fall 2024 Pre-Course Experience Survey
    * CS2 Fall 2024 Post-Course Experience Survey
  - **Spring 2025**
    * CS1 & CS2 Spring 2025 Informed Consent Form
    * CS1 & CS2 Spring 2025 Lab Report
    * CS2 Spring 2025 Pre-Course Experience Survey
    * CS2 Spring 2025 Post-Course Experience Survey
    * CS2 Spring 2025 Program Report
  - **Fall 2025**
    * CS1 & CS2 Fall 2025 Informed Consent Form

    - CS1 & CS2 Fall 2025 Lab Report
    - CS1 & CS2 Fall 2025 Program Report
    - CS2 Fall 2025 Pre-Course Experience Survey
    - CS2 Fall 2025 Post-Course Experience Survey
    - CS2 Fall 2025 Pre-Quiz on Parallelism
  - **Spring 2026**
    - CS1 & CS2 Spring 2026 Informed Consent Form
    - CS1 & CS2 Spring 2026 Lab Report
    - CS1 & CS2 Spring 2026 Program Report
    - CS2 Spring 2026 Pre-Course Experience Survey
    - CS2 Spring 2026 Post-Course Experience Survey
    - CS2 Spring 2026 Pre-Quiz on Parallelism

  The course-wide pre- and post-course survey results are discussed in Section 11.4.1. Results for traditional CS2 concepts are reported in Table 11.11, and results for PDC concepts are reported in Table 11.12.
- **Optional Archive of Syllabi and Course Schedules**
  The `Optional_Archive-Syllabi & Course Schedules` directory contains earlier syllabi and master schedules that support comparison across implementation semesters:
  - CSC 1310 Spring 2025 Master Schedule
  - CSC 1310-001 Spring 2025 Syllabus
  - CSC 1310 Fall 2025 Master Schedule
  - CSC 1310 Fall 2025 Syllabus
  - CSC 1310 Spring 2026 Master Schedule
  - CSC 1310-001 Spring 2026 Syllabus

  These archived files allow instructors and researchers to compare the organization of the course across semesters and to examine how the PDC content was incorporated into the existing CS2 curriculum.

Together, these materials provide the instructional resources necessary to reproduce or adapt the TNTECH CS2 implementation, as well as the course documentation and evaluation instruments needed to examine how the intervention was incorporated and assessed.

Survey and Other Anonymized Data and Analysis

Survey instruments and other evaluation materials are available in the `Evaluation_Data_and_Analysis-Survey_Instruments` directory:
https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter11-TTU/Evaluation_Data_and_Analysis-Survey_Instruments

Data and analysis related to the PDC-infused sorting assignment are discussed in Section 11.3.1, with student assignment feedback summarized in Table 11.9.

Course-wide pre-course and post-course survey results are reported in Section 11.4.1. Results for students' reported experience with core CS2 topics are shown in Table 11.11,

and results for parallel processing, distributed computing, and event handling are shown in Table 11.12.

Chapter 12

CS2 Course Package – Knox College

Java-Based Liberal-Arts Adaptation using the Fork-Join Framework†

David P. Bunde[1] and Jaime Spacco[2]

[1]Knox College, dbunde@knox.edu
[2]Knox College, jspacco@knox.edu

Chapter contents

12.1 Introduction . . . 347
12.2 How CS2 was changed and PDC topics integrated . . . 347
12.2.1 Course calendar . . . 349
12.3 Introducing the Fork-join framework . . . 351
12.3.1 Activity Summary . . . 351
12.3.2 Challenges . . . 351
12.3.3 Data and Analysis . . . 351
12.4 Course-wide Pre and Post Course Surveys . . . 352
12.5 Discussion . . . 354
Appendix . . . 355

Chapter tables

12.1 Knox Updated CS2 Course Calendar . . . 350
12.2 CS2 Pre/Post Survey Analysis Results (Winter 2025) . . . 352
12.3 CS2 Pre/Post Survey Analysis Results (Winter 2026) . . . 353
12.4 CS2 Pre/Post Computing Attitudes Analysis Results (Winter 2026) . . . 354

Abstract

At Knox College, CS2 is where we begin having students write actual parallel code. As the programming paradigm, we chose Java's fork-join framework. This fits with the rest of the course both because the course is taught in Java and because it comes with a programming analysis framework in terms of work (the total amount done by a program) and span (the length of the program's critical path) that fits well with asymptotic analysis as used in the rest of the course. In addition to the programming itself, we explore race conditions and dependencies through an introduction to dependency graphs. Currently, this material fits within a couple of lectures, though we plan to move the introduction of parallel computing earlier in the term so that it can be applied to other programming assignments throughout the term.

Descriptor Elements

Textbook Used: None.

Programming Language Employed: Java

Relevant core courses: CS2/Data Structures

Pre-requisites: Students are assumed to have had one previous programming course in Java (CS1). Transfer students sometimes also take the course after taking CS1 in another programming language and this can work provided that they quickly come up to speed on Java. (The early course assignments have an element of review.)

Relevant PDC topics: Below is a classification of PDC Topics covered in this course according to Bloom's Taxonomy[1]: Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR). Because the course only incorporates a single PDC-related activity, the coverage is broken out base do the parts of that activity as described in §10.4.2.5: the introductory lecture, the prime-counting lab (called lab 8 in the given materials), the reduction lab (called lab 9), and the heat diffusion lab.

PDC Concept	Activity component			
	Lecture	Prime lab	Reduce lab	Heat diff. lab
Task/thread spawning	AP	CR	CR	CR
Notation: library	UN	AP	AP	AP
Dependencies	AN			
Task graphs	CR			
Dynamic task scheduling	AP			
Work	AN			
Span	AN			
Load balancing	UN			
Parallel divide and conquer	AP	CR	CR	CR
Reduction			CR	
Scan			AP	

Learning outcomes:

1. Write parallel programs using Java's fork-join framework,
2. Justify the use of overdecomposition and a cutoff value to end recursion,
3. Define the work and span of a directed acyclic graph and explain their importance, and
4. Define the parallel concept reduction, give some examples, and implement them in Java using the fork-join framework.

Context for use: Knox College is a small, residential liberal arts institution. It uses an academic calendar with 3 terms (Fall, Winter, and Spring), each slightly less than 10 weeks long. The CS2 class meets 4 times a week, 3 lectures (MWF) and a lab on either Tuesday or Thursday (which depends on the term, but all labs are on the same day each term). All class meetings are 70 minutes long. The intent is that a Knox course should be equivalent to a semester course at other institutions.

The size of this class varies, but the targeted size is 24 students. The class is the second required course in the Computer Science major and is also required for the Computer Science minor and the Data Science major. Historically, roughly half of the students continue taking CS courses after this course and complete one of these. The remaining students use the course as an elective and come from a wide variety of majors.

This course uses Peer Instruction [47, 49], a pedagogy where lecture is periodically interrupted with a multiple-choice question. The students answer these individually, then discuss them in groups and re-answer. Then the instructor reveals the correct answer, has a student explain this answer, and then fills in any additional material. The in-class questions are graded only for participation. Lab assignments aren't graded, but are used as a time for students to practice the material with the instructor and a TA available to answer questions. (Variations of some lab problems are assigned as homework, which is graded.)

12.1 Introduction

Our goal with this course was a general modernization while maintaining the flavor of a data structures course. The biggest change was the introduction of the fork-join framework for writing parallel programs. In addition to this, however, we also slightly modernized the course by increasing the focus on practical problems, doing more with graph problems, and switching the exam format.

12.2 How CS2 was changed and PDC topics integrated

Expressing parallelism. The course has long introduced parallelism, using Java threads as the implementation approach. The big change was replacing this with use of the fork-join framework. This is a slightly higher-level abstraction and the notational lift is somewhat higher; introducing threads requires creating a `Runnable` object to represent the work for a thread whereas fork-join embeds the work into a `RecursiveTask` and also submitting to a thread pool. On the other hand, fork-join is more flexible in terms of working for different numbers of processing elements and the associated analysis in terms of work and span fits in with the rest of the course. (You can talk about these when using threads as well, but the number of cores becomes an additional parameter.)

More detail on the use of fork-join is provided in Section 12.3.

More coverage of useful APIs. Part of the modernization push for CS2 at Knox was to have students use more of the standard library. One addition was `Iterator`, which represents an in-progress traversal of a list or other container. Previously, this class had been mentioned, but all the student code using lists involved iterations within the implementation. The `Iterator` class provides a way to traverse and modify list contents while still using the built-in list implementations. This was emphasized to the students by changing one of the list labs into a lab that explicitly uses iterators.

Another API we now discuss is `StringBuilder` (`java.lang.StringBuilder`). As initial motivation, I present the `doubleChar` problem from Codingbat [45], which asks for a function that takes a string and returns a version with each character doubled. For example, `doubleChar("hi")` returns `"hhii"`. The following is a solution that I've shown students in CS1 and which is consistent with the hints given on the Codingbat website:

```
String doubleChar(String s) {
    String retVal = "";
    for(int i=0; i < s.length(); i++) {
        char c = s.charAt(i);
        retVal = retVal + c + c;
    }
    return retVal;
}
```

This discussion comes right after the introduction of Big-O notation so we talk about the representation of Java strings and the running times of string operations; `length` and

`charAt` are constant time, but + requires the creation of a new string and is thus $O(n)$. Therefore, the running time of this implementation of `doubleChar` is $O(n^2)$ because the innocuous 5th line (`retVal = retVal + c + c;`) takes linear time. `StringBuilder` is the Java way around this; it is basically an array-based list of characters with extra space so that additions to the end of the string can be completed easily (constant amortized time if characters are added one at a time). Thus, the very similar `StringBuilder`-based implementation of `doubleChar` runs in $O(n)$ time.

`StringBuilder` is introduced shortly after the introduction of Big-O notation and serves as another opportunity to practice that concept. In particular, it illustrates that code's running time depends on the functions that it uses. It also introduces a library with practical uses; `doubleChar` is not such a use, but we talk about text applications where the output is build up from multiple operations (e.g. chat programs, webpage creation, etc).

Graphs. Another modernization move was to increase the coverage of graphs. The course has long introduced graph terminology and algorithms near the end of the course while also using some of those concepts earlier. For example, one of the labs is based on an EngageCSEdu activity [46] in which students explore a 2D grid "cave" using depth-first search. This is presented several weeks before graphs because it is presented as backtracking and an application of the Stack ADT; the words "graph" and "depth-first search" 'are never used. Another lab talks about the Facebook "friendship graph" and has students build the adjacency list representation of a small graph using `Person` objects, each of which has a list of names of their friends. The application is presented as building a "Friendship graph", but graphs are not explicitly defined, the word is not used other than once in that phrase, and no other terminology (vertex, edge, adjacency list) is used. The lab's apparent topic is creating objects and using lists and maps (a map is used to look up `Person` objects by name). The idea behind both of these labs is to "plant seeds" about the idea of graphs while teaching them simpler concepts. Then, at the end of the course when graphs are formally introduced, these are used as examples of the graphs and their algorithms.

The overall strategy of introducing graph problems before the specific terminology is continued in the new course, as are the existing labs. The content was expanded by introducing the idea of traversing a component so that components could be counted and their sizes identified. This was added to the friendship graph lab; instead of just building the graph and enumerating one component, students are now asked to find the size of the largest one. Students were also asked to traverse components in a 2D grid application where each location has a specific kind of terrain. (Sort of a generalization of cave application except that locations weren't just "wall" or "open"; the stated motivation was the board game Cartographers [71].) Specifically, students were looking for the largest group of connected locations sharing a terrain type or the types of terrain bordering such a connected group of locations. This not a huge change, but does add another graph algorithm as well as another application for graphs to further demonstrate their generality.

Reduction of time on less practical topics. To make time for the added content, some examples were removed or deemphasized. The largest of these is reverse Polish notation, which had previously been covered over the bulk of a lecture. The purpose of this was to

illustrate stacks and give students practice using them. Although the topic was popular with some students, it is fairly niche and stacks are used several times in the graph material (backtracking and finding connected components).

Some time was also saved by reducing the time spent implementing linear data structures (using arrays and linked memory to implement lists, stacks, and queues). This material is kept because it plays a number of important roles in the course:

- Having two concrete implementations of the data structures emphasizes the abstract side of abstract data types (ADTs).
- They also provide motivation for looking at running time and practice analyzing asymptotic running time.
- A basic understanding of the implementations is crucial for understanding the running time of programs using these fundamental data structures.
- Working with linked lists helps students gain proficiency with linked memory.

Despite these benefits, doing the specific implementations was felt to be less of a key programming skill: In C, the only way to get a linked list is to write one, but in newer languages they are generally part of standard libraries (C++, Java) or built into the language itself (Python). These factors led to a compromise: the course retains the clicker questions about this material (ensure students can recognize code for the implementations) and still implement some of the methods, but are assigned fewer of them.

Recursion received a similar treatment. To practice recursion, the course asks them to write recursive versions of linked list methods, similar to how these methods would be covered in a functional language. This example is still covered, preserving the clicker questions about it, but requiring fewer implementations from students. Instead, the topic is covered in actual functional languages in the programming languages class. This compromise still provides some clean examples of recursion, but also bows to the reality that the lack of tail recursion in Java makes these methods impractical.

Exams. In addition to the content changes, the course switched to in-person written exams from take-home exams due to the ability of generative AI systems to solve many of the course's written problems. This required that exams use shorter problems, particularly reducing the amount of programming that students are expected to do.

12.2.1 Course calendar

The new course calendar at a week-by-week level is shown in Table 12.1. Day-by-day level detail as well as the specific lecture materials and assignments are provided in this chapter's GitHub repository.

Table 12.1: Knox Updated CS2 Course Calendar

Week	Topics	Changes
1	Course introduction; Java review; ADTs; implementing array-based lists	
2	Automated testing; JUnit; writing test cases; OO philosophy; Java interfaces	
3	Linked lists; Iterator ADT	More coverage of Iterator, added lab §12.2
4	Running time; Big-O notation; Stack and Queue ADTs and implementations; StringBuilder; design problems	Less time on implementations, added StringBuilder §12.2
5	Midterm exam; Sorting algorithms (selection, insertion, bubble, quick); comparators; logarithms; backtracking	Added backtracking §12.2
6	Map and Set ADTs; binary search; trees; binary search trees; tree traversals	Added connected components HW §12.2
7	Recursion for subset enumeration; recursive linked list operations; parallel computing; beginning of fork-join framework	Less use of recursion §12.2; Fork-join §10.4.2
8	Work and span metrics; Priority Queue ADT; heaps; bucket and radix sorting	
9	Hashing; design problem practice; reductions; graphs; graph traversal	
10	Review for final	

12.3 Introducing the Fork-join framework

12.3.1 Activity Summary

The fork-join framework is introduced exactly as outlined in §10.4.2 since the module was designed specifically for this course. First, a lecture motivates parallel computing, introduces dependency graphs, presents code in terms of the made up keywords `fork` and `join`, analyzes that code in terms of work and span, and then reveals actual Java code for using fork-join without the hypothetical keywords. Then, the students do a lab where they apply the fork-join framework to the problem of counting prime numbers, which is a slight variation of summing an array, the motivating example used to present fork-join. This is discussed in class and they complete the prime counting assignment for homework. Then, the next week, the students complete a lab introducing the generalization of a reduction and implementing some examples.

This material was placed near the end of the course, with the two lab assignments being the last ones of the term. One of the last homework assignments asked them to implement a 2D generalization summing an array (i.e. to sum a 2D array) using fork-join. On the final, they were asked to take an array of small integers (each in the range 0–3) and compute a histogram showing the frequency of each value. This is a reduction and it was one of a list of examples mentioned in lecture, but it was not mentioned in the lab and they were never given code for it.

12.3.2 Challenges

The biggest weakness we see in this activity is that the examples are basically "toy" problems rather than real applications. We did this so that students could first use the fork-join framework in simple settings, without a lot of other code to distract them from the new part. Ideally, this would be followed up with homework that asks them to apply it to a more interesting problem. Using the heat diffusion "alternate lab" discussed in §10.4.2.5 would be a step in this direction. Another idea would be to move this material earlier in the term so that other assignments could also ask for parallel implementations. The challenge with this is ensuring that students are ready for this material. The formal prerequisites (discussed in §10.4.2.5) are recursion and familiarity with object-oriented programming, but the overall fork-join approach is also fairly sophisticated, requiring them to put these ideas together and also to think about objects representing units of work rather than just data.

12.3.3 Data and Analysis

We did not gather data separately on this unit of the course. Thus, the evaluation is rolled into the course-level evaluation (next).

12.4 Course-wide Pre and Post Course Surveys

In the Winter 2025 and Winter 2026 terms, we conducted pre and post surveys in our CS2 course. We asked students their familiarity with key data structures, broader CS2 topics, as well as parallel and distributed computing topics. Similar to our CS1 course, we conducted the pre-survey in the first week of the term, and the post-survey in the last week. In Winter 2025, a total of 22 students took both the pre and post-surveys, while in Winter 2026, 17 students took both surveys. Questions were scored on a 5-point Likert scale ("Not comfortable at all", "Slightly comfortable", "Moderately comfortable", "Very comfortable" and "Extremely comfortable"). We test each item using the Wilcoxon signed-rank test on matched pairs and report effect sizes as Cliff's delta. Thus, p-values are exploratory/unadjusted. Our results for Winter 2025 are presented in Table 12.2 and for Winter 2026 are in Table 12.3.

Table 12.2: CS2 Pre/Post Survey Analysis Results (Winter 2025)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Data Structures and Algorithms									
Classes/Objects	22	3.41	4.14	0.73	14.0	0.005	Yes	0.43	medium
Lists	22	3.09	4.23	1.14	2.5	0.001	Yes	0.57	large
Generics	22	2.45	3.86	1.41	0.0	0.000	Yes	0.66	large
Stacks	22	2.00	3.91	1.91	0.0	0.000	Yes	0.77	large
Queues	22	2.05	3.91	1.86	0.0	0.000	Yes	0.76	large
Sorting Algos	22	2.36	3.73	1.36	3.5	0.000	Yes	0.61	large
Comparators	22	1.95	3.64	1.68	0.0	0.000	Yes	0.73	large
Recursion	22	2.45	3.77	1.32	5.0	0.000	Yes	0.58	large
Maps	22	1.77	3.73	1.95	0.0	0.000	Yes	0.80	large
Sets	22	2.00	3.68	1.68	0.0	0.000	Yes	0.71	large
Trees	22	1.73	3.86	2.14	0.0	0.000	Yes	0.85	large
BST	22	1.73	3.82	2.09	0.0	0.000	Yes	0.83	large
Priority Queues	22	1.59	3.55	1.95	0.0	0.000	Yes	0.80	large
Hashing	22	1.86	3.68	1.82	0.0	0.000	Yes	0.75	large
Graphs	22	1.41	3.09	1.68	0.0	0.000	Yes	0.73	large
PDC Topics									
Parallel	22	2.14	2.68	0.55	14.0	0.016	Yes	0.30	small
Distributed	22	1.86	2.50	0.64	13.5	0.009	Yes	0.33	medium
Event	22	2.55	3.09	0.55	18.0	0.027	Yes	0.29	small
Programming Skills									
Problem Decomposition	22	3.23	4.00	0.77	7.0	0.002	Yes	0.44	medium
Unit Tests	22	2.50	3.91	1.41	0.0	0.000	Yes	0.66	large
Big-O	22	2.00	3.77	1.77	0.0	0.000	Yes	0.78	large
Data Structure Choices	22	2.55	3.91	1.36	0.0	0.000	Yes	0.65	large
Designing/Writing Classes	22	3.00	4.09	1.09	4.5	0.000	Yes	0.54	large

We observe significant improvements for all data structure and algorithm topics for both courses. For students' comfort level with PDC topics (parallel, distributed, event-driven), we see small or medium improvement in the Winter 2025 term, then medium or large improvement in the Winter 2026 term. The reason for the change is unclear, but it could be that the instructor's own comfort level with the new material allowed them to frame the material in a more coherent way for students. We also see medium or large improvements in core CS2 programming skills across both terms.

Table 12.3: CS2 Pre/Post Survey Analysis Results (Winter 2026)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Data Structures and Algorithms									
Classes/Objects	17	4.88	5.71	0.82	7.0	0.007	Yes	0.42	medium
Lists	17	4.29	5.41	1.12	3.0	0.002	Yes	0.48	large
Generics	17	2.76	4.24	1.47	0.0	0.001	Yes	0.57	large
Stacks	17	2.53	4.88	2.35	0.0	0.000	Yes	0.77	large
Queues	17	2.53	5.06	2.53	0.0	0.000	Yes	0.80	large
Sorting Algos	17	3.00	4.88	1.88	0.0	0.000	Yes	0.66	large
Comparators	17	2.88	4.94	2.06	0.0	0.000	Yes	0.69	large
Recursion	17	3.47	4.94	1.47	0.0	0.001	Yes	0.54	large
Maps	17	2.82	4.94	2.12	0.0	0.000	Yes	0.70	large
Sets	17	2.82	4.94	2.12	0.0	0.000	Yes	0.71	large
Trees	17	2.59	5.18	2.59	0.0	0.000	Yes	0.82	large
BST	17	2.59	5.06	2.47	0.0	0.000	Yes	0.79	large
Priority Queues	17	2.53	4.65	2.12	0.0	0.000	Yes	0.71	large
Hashing	17	2.88	4.71	1.82	0.0	0.000	Yes	0.62	large
Graphs	17	2.18	4.35	2.18	0.0	0.000	Yes	0.74	large
PDC Concepts									
Parallel	17	2.35	3.59	1.24	0.0	0.001	Yes	0.50	large
Distributed	17	2.35	3.29	0.94	2.0	0.003	Yes	0.39	medium
Event	17	3.00	3.94	0.94	8.0	0.008	Yes	0.34	medium
Programming Skills									
Unit Tests	17	2.82	5.12	2.29	0.0	0.000	Yes	0.72	large
Big-O	17	2.59	4.94	2.35	0.0	0.000	Yes	0.74	large
Data Structure Choices	17	2.82	4.88	2.06	0.0	0.000	Yes	0.69	large
Designing/Writing Classes	17	4.53	5.35	0.82	6.0	0.005	Yes	0.38	medium
Problem Decomposition	17	3.24	4.71	1.47	0.0	0.001	Yes	0.53	large

For the Winter 2026 term, our pre and post survey also asked about students' attitudes towards and understanding of a variety of topics, including parallel and distributed computing. These results are presented in Table 12.4. This part of the survey used a 7-point Likert scale. Students report a moderate and significant improvement in their level of computing understanding, and a substantial and significant improvement to their PDC understanding after taking the course. They did not see a significant change in their self-reported level of computing interest, PDC interest, or their perceived importance of PDC topics. This suggests that the new course design is not decreasing student interest in computing, and is increasing their understanding of PDC and computing topics. However, students did not report increased interest in PDC or attach greater importance to PDC. Overall, these are promising results.

Table 12.4: CS2 Pre/Post Computing Attitudes Analysis Results (Winter 2026)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Computing Attitudes									
computing interest	17	5.88	5.82	-0.06	11.5	0.779	No	−0.04	negligible
PDC interest	17	5.06	5.41	0.35	10.0	0.174	No	0.18	small
PDC importance	17	5.82	6.06	0.24	10.5	0.285	No	0.14	negligible
computing understanding	17	4.71	5.71	1.00	1.5	0.003	Yes	0.46	medium
PDC understanding	17	3.41	4.88	1.47	3.5	0.002	Yes	0.48	large

12.5 Discussion

We have successfully taught this course and are comfortable sharing it, but we plan to continue making changes. In particular, we plan on introducing parallelism earlier in the term so that the students can use the fork-join framework on larger programs. One candidate assignment is a game tree exploration, which allows easy parallelism (with some duplicated work when transpositions of the moves yield identical positions) and a problem instance large enough for parallelism to provide a noticeable improvement in running time.

Appendix

GitHub Link to Instructional Material

The materials for this chapter are available in our GitHub repository.

Detailed Pre and Post Syllabus

Also included are a course syllabus and a day-by-day summary of what was taught (with links to the materials used and the changes highlighted in **bold**).

Organization of Material

The GitHub repository contains the full course with the modifications described in this chapter (as taught in Spring 2025). `COURSE_CALENDAR.md` depicts the day by day presentation of material and due dates for assignments. This document has links to the actual materials, which are stored in subdirectories of the `Archive` directory. (The categories are lectures, labs, homework, and exams.) The best way to navigate the material is likely to be through the links in the course calendar, though the archive's organization by type is useful for instructors only interested in a particular type of material. The directory `Detailed_Pre_and_Post_Syllabus` contains the course syllabus.

An updated version of the materials for the fork-join activity are included in the directory `Plugged_Activities/ForkJoin`. This contains the following based on the Knox presentation of this activity by David Bunde:

- The slides used to teach this material with notes added to indicate what he says on each.
- Links to two videos (1, 2) of David lecturing using this material (but using slightly older slides). These are not intended for dissemination to students, but are provided for instructor reference.
- Three lab assignments that students can use to practice the material. David has used two of these in class; the third is provided as an alternative (the main labs are pretty minimalist, but the third is a more interesting application and requires the students to fill in part of a given program).

Materials related to the other changes discussed in this chapter are included as well:

- **Iterator.** The lecture introducing iterators is in LinkedList2.pdf (slides 8–11) and then they are used heavily in Lab 3 on finding common queries (handout, given code).
- **StringBuilder.** The lecture using StringBuilder is StringBuilder.pdf.
- **Graphs.** The early introduction of graph material was in Lab 5 on backtracking to explore a cave (handout, given code) and Lab 6 on building the "friendship graph" (handout, given code). They also practice it in the connected component part of Homework 6 (handout, given code).

Survey and Other Anonymized Data and Analysis

Student-level survey data are not publicly released, but aggregate results are reported in the chapter.

Chapter 13

CS2 Course Package – University of Southern Indiana (USI)

Structured Java Infusion through Event-Driven Programming, Fork–Join, and Parallel Image Processing[†]

Srishti Srivastava

University of Southern Indiana

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

13.1 Introduction . . . 364
13.2 How CS2 was changed and PDC topics integrated . . . 365
13.2.1 Integrated Syllabi (Pre and Post) . . . 365
13.2.2 Highlights . . . 365
13.2.3 Challenges . . . 365
13.3 New Unplugged Activities . . . 368
13.3.1 The Overly Complicated Machine (OCM) . . . 368
13.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 368
13.3.1.2 Implementation Details . . . 369
13.3.1.3 Experience and Teaching Tips . . . 371
13.3.1.4 Data and Analysis . . . 371
13.3.2 Fork Join on Post-its . . . 372
13.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 372
13.3.2.2 Implementation Details . . . 373
13.3.2.3 Experience and Teaching Tips . . . 375
13.3.2.4 Data and Analysis . . . 375
13.4 New Plugged-in Activities . . . 377
13.4.1 Greenfoot Plugged-In Activity . . . 377
13.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 377
13.4.1.2 Algorithms . . . 378
13.4.1.3 Implementation Details . . . 379
13.4.1.4 Experience and Teaching Tips . . . 379
13.4.1.5 Data and Analysis . . . 380
13.4.2 Fork-Join Parallel Sum . . . 382
13.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 382
13.4.2.2 Algorithms . . . 383
13.4.2.3 Implementation Details . . . 383
13.4.2.4 Experience and Teaching Tips . . . 384
13.4.2.5 Data and Analysis . . . 385
13.4.3 Seeing Concurrency: A Java-Based Parallel Image Processing . . . 386
13.4.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 386
13.4.3.2 Algorithms . . . 387
13.4.3.3 Implementation Details . . . 388
13.4.3.4 Experience and Teaching Tips . . . 390
13.4.3.5 Data and Analysis . . . 390
13.5 Course-wide Pre and Post Course Surveys . . . 392
13.5.1 Results and Analysis : Spring 2025 Survey Data . . . 392
13.5.2 Results and Analysis : Spring 2026 Survey Data (PDC Intervention) . . . 395
13.5.3 Did the PDC Intervention Improve Learning? . . . 397
13.6 Other Activities and Ideas . . . 399
13.7 Conclusion . . . 399
13.7.1 Future Work . . . 400
Appendix . . . 401

Chapter figures

13.1 A sample "overly complicated machine." Each stage triggers the next, mirroring how one event leads to a response in an event-driven program. . . . 370
13.2 An illustration of fork-join example from the lecture slide deck for the unplugged fork-join activity. . . . 372
13.3 Pre/post analysis of the free-response answers. . . . 377
13.4 An example Greenfoot environment of the Zelda game. . . . 378
13.5 An example input image used for the parallel image processing activity. . . . 389
13.6 Pre to Post shift for every survey item (n = 12), grouped by construct. Green = significant increase ($p < .05$); grey = non-significant. . . . 394
13.7 Themes in open-text learning goals, pre vs post (a student may express more than one theme) . . . 395
13.8 Pre to post shift for every survey item (n = 18). Mean change per item with Cliff's delta effect size. Green bars are significant ($p < 0.05$); grey are not. . . . 398

Chapter tables

13.1 USI CS2 Relevant PDC Topics and Blooms Levels . . . 361
13.2 USI CS2 (CS 358) Class Standing . . . 362
13.3 USI CS2 (CS 358) Declared Majors . . . 362
13.4 USI CS2 (CS 358) General Computing Experience . . . 363
13.5 USI CS2 Spring 2026 Course Syllabus Calendar with Highlighted PDC Content . . . 366
13.6 Approximate Timing of the Overly Complicated Machine Activity . . . 369
13.7 Approximate Timing of the Fork Join on Post-its Activity . . . 374
13.8 Coding scheme for the free-response answers from the pre- and post-tests. . . . 376
13.9 Approximate Timing for the Greenfoot Plugged-In Activity Implemented as a Week-Long Project . . . 380
13.10 Per-team sentiment on the three reflective questions. + = positive; ∼ = mixed; ∘ = not addressed. . . . 381
13.11 Event-handling constructs reported per team (✓ ēxplicitly described in the write-up). 381
13.12 Approximate Timing for the Fork-Join Plugged-In Activity Implemented Part A of a Week Long Project . . . 384
13.13 Quantitative claims reported by each team. "Sweet-spot threshold" is the cutoff each team identified as best. The verified speedup re-computes $T_{\text{serial}}/T_{\text{parallel}}$ from the team's own timings. . . . 385
13.14 Concept coverage across teams. yes for present, pt for partial, no for absent. G1–G6 are defined in the text. . . . 386
13.15 Approximate Timing for the Parallel Image Processing Plugged-In Activity . . . 389
13.16 Consolidated Per-Team Results for the Parallel Image Processing Activity . . . 390
13.17 USI CS2 Spring 2025 Pre/Post Survey Results, Baseline (No PDC Intervention) . . . 393
13.18 USI CS358 (CS2) Pre/Post Survey Results, Spring 2026 . . . 396
13.19 PDC competency items DiD analysis: baseline → intervention semester. . . . 397
13.20 Unprompted PDC mentions in the open ended question (Q2). . . . 399

Abstract

The ubiquitous nature of parallel and distributed computing (PDC) in modern software systems requires an early and deliberate introduction to these concepts within the undergraduate computer science curriculum. While foundational tracks historically isolate students within a strict sequential programming paradigm, this chapter presents a successful course exemplar from the University of Southern Indiana (USI) that modernizes the CS2 course, "Intermediate Object-Oriented Programming", by integrating PDC principles directly into core topics. This curriculum redesign incrementally introduces students to concurrent thinking through a balanced mix of unplugged activities and plugged-in programming modules. As students master intermediate Java concepts, they simultaneously explore data parallelism, event-driven programming, and parallel processing via Java multithreading. This chapter outlines the pedagogical motivations, architectural course changes, and design decisions behind the transformation. It provides a detailed look at how active learning strategies and hands-on, project-based assignments allow students to absorb complex, concurrent concepts in an accessible, highly contextualized environment. Moreover, the paper offers insights into student engagement, concept retention, and practical programming competency, providing actionable recommendations for educators looking to foster a parallel computing mindset in early CS courses.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Starting Out With Java: Control Structures through Objects, 8th edition, Pearson Revel 2022, by Tony Gaddis.

Relevant core courses: CS2

Pre-requisites: At USI, students enrolling in the CS2 course should have completed their CS1 course with a minimum of D grade or higher. They should have also completed a discrete math course with a minimum of D grade or higher.

Relevant PDC topics: Table 13.1 lists the integrated PDC topics with bloom's classification (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV), and Creating (CR)).

Learning outcomes from PDC interventions: Upon completing the PDC intervention, students are expected to be able to:

- **LO1 — Data and task parallelism.** Recognize and define parallelism as multiple operations executing at the same time; explain the shift from the sequential "only one thing happens at a time" assumption to a parallel decomposition; implement a divide-and-conquer decomposition that splits a problem (summing an array, counting primes, or filtering every pixel of an image) into independent subtasks; analyze how a task tree divides and recombines work; and evaluate the resulting parallel behavior against a serial baseline.

Table 13.1: USI CS2 Relevant PDC Topics and Blooms Levels

PDC Concept	Chapter Section				
	Overly Complicated Machine (13.3.1)	Fork/Join on Post-its (13.3.2)	Greenfoot (13.4.1)	Fork-Join Parallel Sum (13.4.2)	Seeing Concurrency (13.4.3)
Data/Task Parallelism	RE, UN	RE, UN, AN		RE, UN, AP, AN, EV	RE, UN, AP, AN, EV
Shared Memory		RE, UN		RE, UN, AP	RE, UN, AP
Scheduling/Load Balancing		RE, UN, AP		RE, UN, AN	RE, UN, AN
Speedup/Amdahl's Law		RE, UN, AN, EV		RE, UN, AP, AN, EV	RE, UN, AP, AN, EV
Event Handling	RE, UN, AP		RE, UN, AP, CR		

- **LO2 — Shared memory and synchronization.** Identify shared, mutable state in a concurrent program; explain how uncoordinated access produces race conditions and lost updates (for example, two threads reading a counter or a bank balance before either writes); and apply a coordination mechanism passing results back on a join.
- **LO3 — Scheduling and load balancing.** Recall that a pool of workers executes submitted tasks; describe how a cutoff (threshold) or a configurable thread count controls the granularity of those tasks; apply that parameter to distribute work across a worker pool; and analyze how the distribution affects processor utilization.
- **LO4 — Speedup and Amdahl's Law.** Define speedup as the ratio of serial to parallel runtime; explain why the ideal linear speedup is rarely achieved; apply the speedup formula to their own measured timings; analyze how the sequential portion of a program and the overhead of creating, switching, and synchronizing threads bound the achievable speedup; and evaluate empirical timing tables to locate the "sweet spot" at which additional tasks or threads stop helping.
- **LO5 — Event-driven programming and event handling.** Define an event and an event handler; explain the event-driven execution model in which code responds to independent, user- or system-generated actions rather than running as a single predetermined sequence; implement event handlers that respond to input such as key presses and collisions; and create an original, meaningfully different interactive program built around event handling.

Context for use: These PDC interventions were implemented in CS 358, *Intermediate Object-Oriented Programming*, the CS2 course in the Computer Science program at the University of Southern Indiana (USI), a mid-sized public regional university in Evansville, Indiana, during the Spring 2025 and Spring 2026 semesters. Students typically enter as second-year (sophomore) and early-junior computer science majors who have completed the introductory programming sequence and have working familiar-

ity with basic control structures, data types, and elementary object-oriented concepts. The section enrolled roughly 20–35 students and meet in a 75-minute face-to-face session two times a week. All programming was done in Java, using Tony Gaddis's *Starting Out with Java: Control Structures through Objects* as the primary text alongside VS Code programming environment. The PDC content was integrated at points where it reinforced the course's object-oriented and event-based objectives rather than taught as a standalone unit. Event-driven programming was introduced through an unplugged event-driven activity and hands-on Greenfoot exercises in the JavaFX and Event Handling module, and shared-memory parallelism was developed in the Multithreading and Recursion module through a Fork/Join "unplugged and plugged-in" activity and work with Java's `ExecutorService` API. This placement let the students encounter concurrency and event-handling concepts as natural extensions of the inheritance, polymorphism, and recursion topics they were already learning.

The participant profile is drawn from the CS 358 pre-course experience surveys administered across two offerings of the course. Spring 2025 served as the baseline semester. In Spring 2026, the PDC interventions were introduced. In each semester, the pre-surveys were completed by 20 students, creating a pooled sample of 40. Neither survey instrument collected gender or race/ethnicity, so those distributions are not reported. Across the two cohorts, participants were spread fairly evenly across class standings. Juniors formed the largest single group (14 of 40, 35.0%), followed closely by first-year students (13, 32.5%) and sophomores (12, 30.0%), with a single senior (2.5%) (Table 13.2). The group was almost entirely composed of Computer Science majors (38 of 40, 95.0%); the remaining two students were one Computer Information Systems major and one Computer Science and Mathematics double major (Table 13.3). Students' self-reported sense of their general computing background was moderately positive and broadly consistent across the two offerings: a majority expressed some level of agreement that they had a good understanding of computing (28 of 40, 70.0%, across "somewhat agree," "agree," and "strongly agree"), while 15.0% were neutral and 15.0% expressed mild disagreement; no student strongly disagreed (Table 13.4).

Table 13.2: USI CS2 (CS 358) Class Standing

Class Standing	Spring 2025 (n=20)	Spring 2026 (n=20)	Total (n=40)
First year (freshman)	6	7	13 (32.5%)
Sophomore	7	5	12 (30.0%)
Junior	6	8	14 (35.0%)
Senior	1	0	1 (2.5%)

Table 13.3: USI CS2 (CS 358) Declared Majors

Declared Major	Spring 2025 (n=20)	Spring 2026 (n=20)	Total (n=40)
Computer Science	19	19	38 (95.0%)
Computer Information Systems	0	1	1 (2.5%)
Computer Science & Math	1	0	1 (2.5%)

Table 13.4: USI CS2 (CS 358) General Computing Experience

Computing Experience	Spring 2025 (n=20)	Spring 2026 (n=20)	Total (n=40)
Strongly disagree	0	0	0 (0.0%)
Disagree	1	0	1 (2.5%)
Somewhat disagree	2	3	5 (12.5%)
Neutral	1	5	6 (15.0%)
Somewhat agree	5	6	11 (27.5%)
Agree	8	2	10 (25.0%)
Strongly agree	3	4	7 (17.5%)

13.1 Introduction

Many instructors teaching CS2 courses face the same challenges that complicate parallel and distributed computing (PDC) integration in CS1, but in a more acute form. By the second semester the course already carries a heavy conceptual load, including topics such as, advanced object-oriented design, inheritance and polymorphism, exception handling, event-driven graphical programming, recursion, and an introduction to multithreading. The prospect of adding PDC content without displacing core learning objectives can feel more daunting than CS1. As a result, deliberate PDC integration in CS2 continues to be the exception rather than the norm. Often it is confined to a single late-term lecture on threads. This chapter is written for instructors who share that concern but are looking for practical, tested strategies to resolve it. As with the preceding CS1 chapter, it is written for instructors at institutions similar to USI, a mid-sized public master's university with a community-engaged mission, where resources are not unlimited, class sizes are modest, and students arrive at CS2 with uneven mastery of the prerequisite CS1 material. The strategies described here do not require additional instructional time beyond a standard 50-minute to 75-minute class session, do not presuppose access to parallel computing hardware or specialized cluster environments, and are designed to complement rather than compete with the existing learning objectives of an intermediate level object-oriented programming course.

Students completing the CS1 chapter's modules will already have encountered a deliberate preview of brief, demonstration-level introduction to event-driven programming through Greenfoot, explicitly framed as a look ahead to the follow-up CS2 course. The modules described in this chapter take up that foundation and formalize it, first turning the event-driven preview into a full treatment of event handling and the concurrency inherent in it, and then extending students from the sequential, divide-and-conquer reasoning they develop through recursion into genuine task and data parallelism. However, the PDC activities described in this chapter can also be adopted for a CS2 course where students do not have any prior PDC experience. The PDC modules use a combination of unplugged and plugged-in activities by first giving students a physical, embodied experience of a concept and then connecting that experience to a programming task that operationalizes the same concept.

The first PDC module addresses event-driven programming and the concurrency implicit in it. It begins with an unplugged activity, "Event Handling Unplugged: An Overly Complicated Machine," in which students draw an event-dispatch system on paper. Further, a plugged-in review of event handling is implemented as a midterm group project built in Greenfoot [34]. The second module introduces task and data parallelism through the fork/join paradigm. It begins with a "Fork/Join Unplugged" activity, in which students participate in a post-it based problem decomposition into independent subproblems and recombine results for generating an overall solution. This is followed by two plugged-in activities in the final project that uses Fork-Join to sum elements of a very large array in parallel and Java's `ExecutorService` API to perform parallel image processing. This module builds directly on the recursion and multithreading content of the standard CS2 syllabus, using the divide-and-conquer structure students already associate with recursion as the entry point to parallel decomposition, speedup, and load balancing.

13.2 How CS2 was changed and PDC topics integrated

A central design principle of the approach described in this chapter, carried over directly from the CS1 chapter, is that PDC concepts can be integrated into a CS2 course without restructuring the syllabus, displacing core content, or requiring additional class sessions. Rather than treating PDC as a standalone topic that competes for time alongside the traditional CS2 curriculum, the integration strategy adopted here embeds PDC thinking directly into the instructional contexts where foundational programming concepts are already being taught. In CS2 this embedding is especially seamless, because the two modules chosen as intervention points, event handling and multithreading, are places where the standard curriculum was already brushing up against parallel and concurrent reasoning. The result is a course that looks, from a structural standpoint, nearly identical to a conventional CS2 offering, but in which selected modules are taught through a lens that naturally introduces parallel reasoning. To integrate the activities and the assignments associated with the new PDC modules, certain JavaFX assignments were eliminated; the textbook's repetitive handling of key JavaFX concepts made it possible to streamline this material without sacrificing core learning outcomes.

13.2.1 Integrated Syllabi (Pre and Post)

The course calendar from the syllabus for CS2 (CS358, Spring 2026) at USI is shown in Table 13.5. It spans sixteen weeks across eight content modules (numbered Module 0 through Module 7). The PDC interventions, highlighted in the course calendar, are integrated into this existing structure at two points: the *JavaFX and event-handling* module and the *multithreading and recursion module*. For the final project, students demonstrate semester long learning of programming concepts along with PDC multithreading concepts in a parallel image processing application.

13.2.2 Highlights

The CS2 course structure remained largely intact, retaining all eight content modules, quizzes, exams, and core programming projects in their original sequence. To integrate the new PDC materials, a small amount of repetitive JavaFX content was trimmed from the calendar without compromising any learning objectives. The PDC content was introduced primarily through additions and repurposings. The Module covering event-handling instruction was augmented with an unplugged activity and a Plugged-in Greenfoot exercise, while the traditional midterm was reframed into a hands-on Greenfoot group project assessing the same core competencies. Additionally, Module 6 integrated the 'Fork/Join Unplugged' activity within its existing timeframe, and Module 7 repurposed the standard final project slot for a parallel image processing deliverable using Java's ExecutorService API.

13.2.3 Challenges

The challenges encountered during PDC intervention in CS2 are described as challenges of adoption, challenges of adaptation, and challenges arising from the research layered on

Table 13.5: USI CS2 Spring 2026 Course Syllabus Calendar with Highlighted PDC Content

Modules	**Week**	**Assigned Reading**	**Assignments**	**Quizzes and Exams**
Module 0: CS258 Recap	Week 1	Read Syllabus, Start Here Content, Instructor Notes	CS358 Pre-Course Experience Survey	Review Quiz
Module 1: A Second Look at Classes and Objects	Weeks 2–3	Chapter 8	Programming Projects 1, 2, 3	Quizzes 8.1, 8.3, 8.4, 8.5
Module 2: Text Processing and Wrapper Classes	Weeks 4–5	Chapter 9	Programming Projects 1, 2, 3	Quizzes 9.2, 9.3, 9.6
Module 3: Inheritance	Weeks 6–7	Chapter 10	Programming Project	Quizzes 10.1, 10.2, 10.3, 10.8, 10.10
Module 4: Exceptions and Advanced File Handling	Week 8	Chapter 11	Programming Projects 1, 2	Quizzes 11.1, 11.3
Week 9: Spring Break				
Module 5: JavaFX and Event Handling	Weeks 10–11	Chapters 12–14; Event Handling Unplugged: An Overly Complicated Machine; Review Event Handling using Greenfoot	Midterm Exam (group project) using Greenfoot	
Module 6: Multithreading and Recursion	Weeks 12–14	Chapter 15; Fork/Join Unplugged	Programming Projects 1, 2	Quizzes 15.1, 15.2
Module 7: Final Project and Review	Weeks 15–16	Final Project Introduction: Fork/Join Plugged-In, ExecutorService API in Java and Parallel Image Processing	CS358 Post-Course Experience Survey; Final Project (Lab Days)	

top of instruction. The CS2 setting adds a more prominent technical dimension, which is folded into the discussion of adaptation below.

Adoption challenges: the most immediate obstacle, was the in-class coordination of the unplugged activities. Both "An Overly Complicated Machine" and the "Fork/Join Unplugged" exercise require physical materials, group formation, and timed phases that must be orchestrated within a fixed class period. For example, the Fork/Join Unplugged activity requires partitioning a task using post-it notes among the members of a group and then coordinating an explicit recombination step. In our sections of 28–34 students meeting in a standard lecture room, forming groups, distributing materials, and reserving time for a substantive post-activity discussion required careful minute-by-minute planning. Instructors adopting these activities should budget preparation time accordingly and should not assume that transitions between phases will absorb themselves into the period.

Adaptation challenges: The activities described in this chapter are calibrated to a particular room, roster size, and staffing level, and the CS2 modules add an extra layer of challenge from the technical dimension. The plugged-in components, event handling in Greenfoot, the Greenfoot midterm group project, and the `ExecutorService`-based parallel image-processing final project, depend on a working development environment, and instructors should verify well before the relevant weeks that Greenfoot is installed and that students' Java environments support the concurrency and image-processing libraries the final project requires. Larger sections would lengthen the group formation step disproportionately, and rooms with fixed furniture make the unplugged enactments more time-consuming than anticipated. Staffing is the other adjustable variable. Having a teaching assistant present to help manage timing, circulate between groups, and troubleshoot environment problems during the Greenfoot and fork/join sessions is strongly advisable for sections larger than 30 students. Instructors should expect to modify phase timings and, for the plugged-in work, the specific tooling, rather than the underlying structure of the activities.

Research challenges: Administering the pre- and post-activity assessments in a way that preserved their research validity without disrupting the flow of the session proved to be a distinct challenge from teaching the activities themselves. The pre-tests had to be completed before the corresponding PDC content was introduced, which constrained the sequencing of the class period and required explaining to students why they were being asked about topics such as concurrency, event-handling, or data parallelism, that they had not yet encountered. Some students found this ambiguity uncomfortable, and a few initially misread a pre-test as a graded quiz. Instructors should state explicitly, both verbally and in writing, that pre-test responses carry no grade consequence and that incorrect answers are expected and informative. Moreover, instructors intending to collect data for publication should leave sufficient buffer time before the semester begins to secure IRB approval at their own institutions.

13.3 New Unplugged Activities

13.3.1 The Overly Complicated Machine (OCM)

Event handling is often a student's first encounter with a control flow that is not strictly sequential. Instead of statements executing in the order they are written, fragments of code lie dormant until some external occurrence such as a click, a keystroke, or a page finishing its load, causes them to activate. This PDC module is an unplugged activity that builds an intuition for this inversion of control. Students are asked to design a whimsical, Rube Goldberg [42] style contraption in which a chain of five "events" cascades toward a final goal, and then map that everyday cause-and-effect chain onto the vocabulary of event-driven programming using events, responses, and event handlers.

13.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

Problem description. This activity introduces the concept of event handling and event-based programming to students. Working in small teams, students brainstorm an "overly complicated machine", in which each step is triggered by the step before it. Each team's machine must contain exactly five distinguishable events and must accomplish some concrete final goal (for example, watering a plant, ringing a bell, or turning off a light). Teams sketch the machine on paper, explicitly labeling each of the five events, and then explain their design to a neighboring team. A guided reflection then draws the analogy between the physical chain reaction the students invented and the way a program reacts to events at run time. Every event in their machine has a corresponding response, just as every event in a graphical or web program must have an event handler that runs in response to it.

Prerequisites. The activity is intentionally lightweight in its technical prerequisites, which is what makes it a suitable introduction to the topic. Students should have completed an introductory programming sequence and be comfortable with basic control flow (sequencing, conditionals, and methods) so that the later contrast with non-sequential, event-driven execution is meaningful. No prior exposure to graphical user interfaces, JavaScript, or Java's event model is required. Because the exercise is unplugged, students need no computers, development environment, or programming language during the activity itself.

Learning outcomes. After completing this activity, students will be able to:

- **Define an event** as an action or occurrence that signals an interaction with, or a change in the state of, a program.
- **Define an event handler** as the code that is written to react to a specific event and produce the appropriate response.
- **Distinguish an event from its response**, recognizing that events and handlers form pairs and that a program must supply a handler for each event it cares about.
- **Explain the event-driven paradigm** at an intuitive level, contrasting the reactive, "wait-then-respond" flow of an event-driven program with the top-to-bottom flow of a sequential program.

- **Reason about many-to-one triggering**, recognizing that a single device (or handler) may be activated by more than one distinct event.

13.3.1.2 Implementation Details

Logistics. The activity is unplugged and requires only inexpensive, readily available materials:

- One sheet of blank paper per team, used for scratch work, brainstorming, and the final labeled drawing of the machine.
- Writing utensils (pens, pencils, or markers); colored markers are helpful but not required.
- A projector or slides to display the prompt, the worked example, and the reflection questions.

Students work in teams of three to four. The activity fits comfortably in a single class meeting and typically takes 30–35 minutes end to end. Table 13.6 shows approximate timing of different phases of the activity. The activity also scales well to large classes, since no equipment beyond paper is needed and teams work independently.

Table 13.6: Approximate Timing of the Overly Complicated Machine Activity

Activity	**Duration**
Welcome and learning goals	2 min
Introduction: motivating event handling and event-driven programming	2 min
Worked example: a Rube Goldberg–style "overly complicated machine"	2 min
Setup: Activity overview, rules (five labeled events, one final goal, names on back), distribution of paper and markers, team formation (3–4 students)	4 min
Design phase: teams brainstorm and draw a machine with five events that accomplishes a final goal	10 min
Peer exchange: teams combine into groups of 6–8 and take turns (∼2 min each) describing their machines	5 min
Guided reflection: hardest part of the design, triggering one device with two events, how the activity mimics events in code	5 min
Formal content: definitions of *event*, *response*, and *event handler*, with paired examples	5 min
Total	**46 min**

How to conduct. The activity proceeds in three phases: design, peer exchange, and reflection.

1. **Frame the task with an example.** Begin by showing an image of an "overly complicated machine": a Rube Goldberg–style contraption in which one action triggers the next. A sample image is shown in Figure 13.1.
2. **Design phase (teams of 3–4).** Ask students to form teams of three to four. One volunteer per team takes the sheet of paper for scratch work and brainstorming. As a team, students design an overly complicated machine with exactly five events that accomplishes some final goal. The machine must be drawn on the paper with each of the five events clearly labeled. Have each team member write their name on the back of the page (useful for participation credit and for keeping teams accountable).

3. **Peer-exchange phase (groups of 6–8).** Once every team has a design, ask each team to pair up with one other team, forming a larger group of six to eight students. Teams take turn describing their overly complicated machine to the other team, walking through the five events in order. Explaining the chain aloud forces students to articulate what triggers each step and what happens in response, which is precisely the event/response structure the lesson will name.
4. **Guided reflection.** Reconvene the class and lead a short discussion using the following prompts:
    - What was the most difficult part of designing an overly complicated machine with five events?
    - What if one of the devices had to be triggered by two events instead of just one? Would that be possible?
    - How does this activity mimic the way events happen in code? Use this question to pivot into the formal content.
5. **Transition to formal content.** With the shared experience in place, introduce the programming vocabulary, anchoring each term in what the students just did.

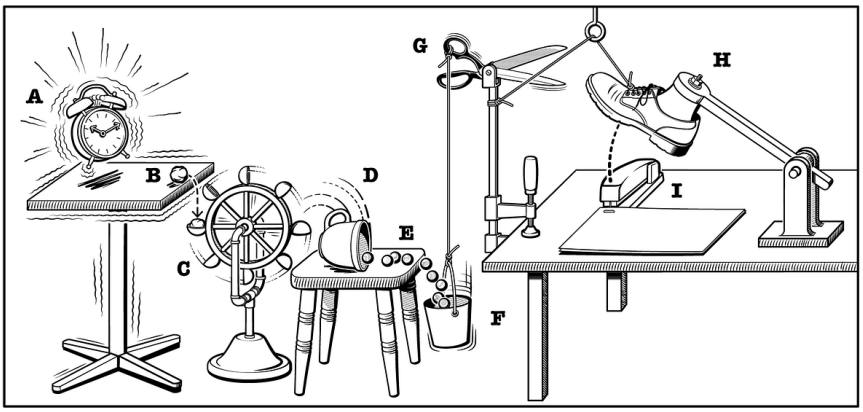


Figure 13.1: A sample "overly complicated machine." Each stage triggers the next, mirroring how one event leads to a response in an event-driven program.

Instructional material. The activity is supported by a short lecture slide deck that (i) presents the prompt and a worked visual example of an overly complicated machine, (ii) states the step-by-step instructions for the design and peer-exchange phases, (iii) poses the reflection questions, and (iv) introduces the formal definitions of *event*, *response*, and *event handler*. The material used for implementing this activity can be found at: USI GitHub OCM. The drawing each team produces doubles as a lightweight, gradable artifact and the sheets can be collected for participation credit. Instructors may extend the activity with a follow-on assignment in which students re-express one of their machine's event chains as pseudocode or, later in the unit, as a small Java program with listeners and handlers, closing

the loop between the unplugged intuition and the plugged-in implementation. At USI, the unplugged activity was further supplemented by a plugged-in Activity using the Greenfoot environment described in Section 13.4.1.

13.3.1.3 Experience and Teaching Tips

The activity works best as a primer delivered immediately before the formal event-handling material rather than as a review. Students who have not yet seen event handlers approach the design task without preconceptions and are then primed to attach the new vocabulary to a concrete, shared experience. The phrase "remember your machine" becomes a durable anchor for the terms *event* and *handler* throughout the unit.

Constraining teams to exactly five events matters. Too few and the chain is too trivial to reveal the idea of triggering; too many and teams spend the whole period negotiating an elaborate contraption instead of reflecting on structure. Five is enough to force students to think about how each step reliably sets off the next event.

Emphasize repeatedly that the drawing "can be ugly." Some students, especially those anxious about art, stall on making the picture look good. Framing the sketch as a communication tool that is good enough to explain to another team keeps the focus on the events and their ordering rather than on presentation.

The peer-exchange phase carries much of the learning. Instructors are advised to circulate during this phase and listen for teams that describe steps in cause-and-effect language as these are ready-made examples to reuse when you introduce handlers. Moreover, students do not automatically see that their paper machine is a model of software behavior. Instructors could state the mapping directly explaining how each labeled step is an event, each next action is a response, and the "code" that would make the response happen is an event handler.

13.3.1.4 Data and Analysis

No formal assessment instrument (such as a validated pre-test, post-test, concept inventory, or engagement survey) was administered for this activity. Instead, its effectiveness was gauged qualitatively through instructor observation during class. The instructor monitored student engagement while teams brainstormed and drew their machines, listened to the explanations exchanged during the peer-sharing phase, and attended to the quality and depth of student responses during the guided reflection. These observations of students' ability to articulate the five events in their designs, to reason about triggering a single device with two events, and to connect their physical machine to the notions of events and event handlers, provided the primary evidence that the activity achieved its intended learning outcomes. Because the data are observational and were not collected with a structured instrument, the findings reported here are formative and anecdotal rather than the result of a controlled study.

13.3.2 Fork Join on Post-its

Before students are introduced to the Java multi-threading APIs, this unplugged activity helps them understand the concept of threads. Working with nothing more than a stack of sticky notes, a class collectively computes the sum of a 32-element array by recursively splitting the work, handing halves to new "worker" threads, and joining partial results back up a tree. The physical act of tearing off a note, passing it to a neighbor, waiting, and then adding two returned numbers makes the otherwise abstract mechanics of *fork*, *join*, and the *sequential cutoff* concrete and memorable. Figure 13.2 shows the example of summing elements of an array that was presented to the students before engaging them in the unplugged activity.

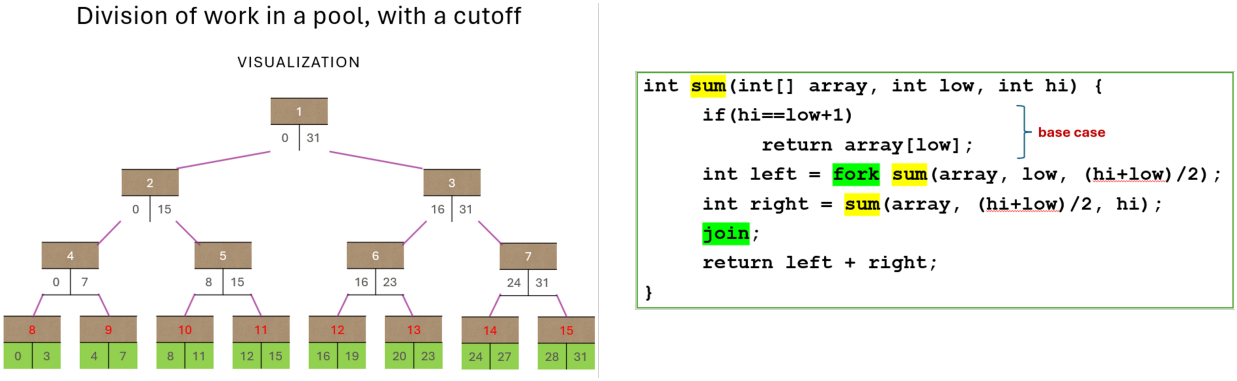


Figure 13.2: An illustration of fork-join example from the lecture slide deck for the unplugged fork-join activity.

13.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

Problem description. The class is given a single array of 32 integers (indices 0–31) and asked to compute its sum. Rather than summing the array one element at a time, students perform the computation the way a fork/join runtime would. A single "root" worker begins with the full range. Any worker holding a range larger than a fixed cutoff (four elements in this activity) does no arithmetic itself; instead it *forks*, splitting its range in half and delegating each half to a new worker on a fresh Post-it note. This division repeats recursively until a range reaches the cutoff, at which point the worker serially sums its four values, writes the partial sum on its note, and *joins* by passing the note back to the parent participant that spawned it. When both of a parent's children have returned, the parent adds the two partial sums and passes the combined result up. The root ultimately reports the total. The activity uses a worker pool of 15, matching the 15 nodes of the resulting binary tree (one root, and 14 descendants down to eight leaf workers).

Prerequisites. Students should be comfortable with:

- basic array indexing and the idea of a half-open or inclusive index range `[low, hi]`;
- writing and tracing a simple sequential loop that accumulates a sum;
- the concept of recursion;

- an intuitive notion that a modern CPU has multiple cores capable of doing work at the same time.

No prior exposure to Java threads, the `Runnable` interface, or the Fork/Join framework is required. This activity can be used as an on-ramp to those topics.

Learning outcomes. After completing the activity, students will be able to:

1. Distinguish sequential from parallel decomposition.
2. Trace the fork/join model and identify which workers divide work versus which perform the base-case computation.
3. Reason about the sequential cutoff.
4. Recognize parallel overhead and idleness and connect this to the real cost of thread creation, coordination, and synchronization.
5. Relate the activity to the idea that the non-parallelizable portion of a task bounds the achievable speedup (Amdahl's Law).

13.3.2.2 Implementation Details

Logistics. The activity is deliberately low-tech and scales to a medium sized classroom.

- **Group size:** A pool of 15 workers matches the tree exactly (1 root, 2 + 4 internal, 8 leaves). With larger classes, run several parallel pools of 15 racing on the same array, or add observers who record timing and idle workers; with smaller classes, let one student play more than one node, or reduce the array to 16 elements (a pool of 7).
- **Materials:**
  - One pad of Post-it notes per pool (roughly 20 notes plus spares).
  - A pre-printed strip showing the 32-element array with its indices 0–31, one copy per pool, taped to a desk or the board.
  - A marker or pen per worker.
  - Optional: a visible timer/clock and a projected copy of the tree diagram.
- **Space:** Room for students to stand and physically hand notes to one another; a whiteboard to reconstruct the tree during the debrief.
- **Parameters (fixed for this run):** array size = 32, sequential cutoff = 4, worker pool = 15.

Table 13.7 lays out how the unplugged activity fitted into a single class period.

How to conduct.

1. **Frame the problem.** Show the 32-element array and ask the class how long a single person would take to sum it one element at a time. Establish that this is the sequential baseline.
2. **Appoint the root.** Give one student the full range $[0, 31]$ written on a Post-it. Announce the rule: *"If your range has more than four numbers, do not add anything. Split your range in half, write each half on a new note, and hand one note to a free worker. Keep the other half yourself as a new, smaller task."*
3. **Fork recursively.** Let the division cascade. The root spawns workers for $[0, 15]$ and $[16, 31]$; those spawn $[0, 7], [8, 15]$ and $[16, 23], [24, 31]$; those in turn spawn the eight

Table 13.7: Approximate Timing of the Fork Join on Post-its Activity

Activity	Duration
Welcome and learning goals	2 min
Introduction: motivating parallelism, sequential vs. parallel work using the word-count example	3 min
Worked example: the array-sum problem and its sequential `simpleSum` baseline	2 min
Setup: Activity overview, rules (split a range if it exceeds the cutoff of four, otherwise sum serially and join), distribution of Post-its and the indexed array strip, worker-pool formation (15 workers)	4 min
Fork (divide) phase: the root worker $[0, 31]$ recursively splits ranges in half, handing new notes to spawned workers until each range holds four elements	6 min
Base-case computation: the eight leaf workers ($[0, 3], \ldots, [28, 31]$) serially add their four values and write the partial sum on their note	7 min
Join (combine) phase: leaves return notes to parents; each parent waits for both children, adds the two partials, and passes the result up to the root	6 min
Guided reflection: idle work-dividing workers, overhead vs. speedup, and the effect of changing the cutoff	5 min
Formal content: definitions of *fork*, *join*, and the *sequential cutoff*, with the bridge to the Java `sum` method and `Thread` class	5 min
Total	**40 min**

leaf ranges of four elements each. Encourage students to physically move and hand off notes so the tree becomes visible in the room.

4. **Hit the cutoff.** When a worker's range holds exactly four elements, they stop splitting, read those four values off the array strip, add them serially, and write the partial sum on their note.
5. **Join.** Each leaf participant hands their note back to the parent who spawned it. A parent may not proceed until both children have returned; it then adds the two partial sums, writes the combined value, and returns it upward. Emphasize the waiting as synchronization.
6. **Report the total.** The root receives two partials, adds them, and announces the array sum. Verify against a sequential sum computed on the side.

Instructional materials. This unplugged activity is designed to sit between two coding artifacts from the lecture, and the following supporting materials accompany it. The instructional material can also be found at the following link: USI GitHub Unplugged Fork-Join.

- **Pre-activity lecture / slides:** the "Multithreading" deck covering sequential vs. parallel programming, the word-count motivating example, parallelism vs. concurrency, processor & thread, overhead, and speedup/Amdahl's Law, ending on the sequential `simpleSum` method (see code snippet below) and the prompt "Let's try this the unplugged way."
- **The tree visualization slides** used to reconstruct and debrief the fork/join structure.
- **Post-activity code walkthrough:** the recursive `sum` method that mirrors the physical protocol, followed by an introduction to the Java `Thread` class and the `Runnable` interface.
- **Optional follow-up assignment:** implement the sequential `simpleSum`, then the recursive `sum`, and finally a threaded version; have students measure and compare running times and relate their results to the idle-worker observation from the activity. For the CS2 course described in this chapter, the follow up assignment was the plugged-

in fork-join activity described later in Section 13.4.2.

The sequential baseline the activity replaces:

```
int simpleSum(int[] array, int low, int hi) {
    int sum = 0;
    for (int i = low; i < hi; i++)
        sum = sum + array[i];
    return sum;
}
```

The recursive divide-and-combine method the activity foreshadows:

```
int sum(int[] array, int low, int hi) {
    int left  = simpleSum(array, low, (hi + low) / 2);
    int right = simpleSum(array, (hi + low) / 2, hi);
    return left + right;
}
```

13.3.2.3 Experience and Teaching Tips

- Let the imbalance land on its own. The most powerful moment is the debrief realization that seven of the fifteen workers did almost no arithmetic; they split a range, handed off two notes, and then waited.
- Make the cutoff tangible. Students often ask why they don't split all the way down to single elements. And ask students to imagine a cutoff of 10,000 and consider how many workers would then be dividing versus computing. This motivates the cutoff as the key that trades coordination overhead against parallelism.
- Emphasize the wait at each join. The visible idling is the intuition behind synchronization cost and why "too many threads can slow a program instead of speeding it up."
- Use two note colors. Distinguishing internal (divide-only) workers from leaf (compute) workers by color makes the tree's structure and the overhead argument becomes self-evident when you rebuild the diagram on the board.
- Verify the answer. Compute the true sum ahead of time (or with a quick sequential pass) so the class can confirm the root's result; a mismatch is a great opening to discuss where a "race" or a missed join could corrupt a parallel computation.
- Close by overlaying the recursive `sum` method on the activity: `left` and `right` are the two child notes, and `return left + right` is the join. This makes the subsequent jump to the `Thread` class and `Runnable` interface feel like naming something students have already physically performed.

13.3.2.4 Data and Analysis

To gauge whether the unplugged activity improved students' grasp of the fork/join model, we administered the same single open-response question immediately before as a pre-test and again after the activity as a post-test in a Spring 2026 offering of the course.

Instrument and design. Both the pre-test and the post-test posed one identical question:

> *How would you change the `simpleSum` method to support parallel execution across multiple threads and CPU cores? Write down your step-by-step approach below to the best of your knowledge.*

Responses were collected anonymously, so the pre- and post-test answers cannot be linked to individual students. The two sets are therefore treated as independent samples rather than matched pairs. The pre-test yielded $n_{\text{pre}} = 17$ responses and the post-test $n_{\text{post}} = 12$.

Coding scheme. Because the collected data are free text, each response was content-coded against seven binary indicators grouped into two composite measures. The first composite captures the conceptual divide-and-combine model independently of any particular vocabulary. The second captures adoption of the specific *fork/join* terminology the activity is designed to instill. Table 13.8 defines the indicators.

Table 13.8: Coding scheme for the free-response answers from the pre- and post-tests.

Indicator	Counted when the response contains...
Decompose	splits the array or work into parts, sections, or chunks.
Parallel	assigns work to multiple threads/cores/processors, or says it runs "in parallel," "simultaneously," or "at the same time."
Combine	adds or combines the partial sums into a final total.
Halve	splits in half / into two equal halves, or splits recursively (binary division).
Cutoff	shows awareness of a threshold below which parallelizing is not worthwhile (the sequential-cutoff idea).
Fork	uses the term *fork* (forked, forking).
Join	uses the term *join* (joined, joining).

Analysis. Figure 13.3 shows the full distributions of the fraction of responses expressing each concept.

The underlying divide-and-combine intuition was already common before the activity: even on the pre-test, 88% of students proposed splitting the array and 53% mentioned distributing the work across threads or cores. The decisive change was in the model and vocabulary that the activity explicitly teaches. Use of the term *fork* rose from 1/17 (6%) to 11/12 (92%) of responses, and *join*, which no student used on the pre-test, appeared in 8/12 (67%) of post-test answers. The complete fork–join model, using both terms together, went from 0/17 to 8/12 students. Recognition of the recursive halving structure also increased from 6% to 42%.

The samples are small ($n = 17$ and 12) and drawn from a single class, so the effect sizes are estimated with wide uncertainty and may not generalize. Because responses were anonymous, the design is between-subjects rather than a matched pre/post comparison, and the two groups may differ for reasons unrelated to the activity. All responses were scored by a single coder against a purpose-built rubric. Therefore, this analysis is presented as encouraging classroom evidence that motivates a larger, matched, and controlled study, rather than as a definitive effect estimate.

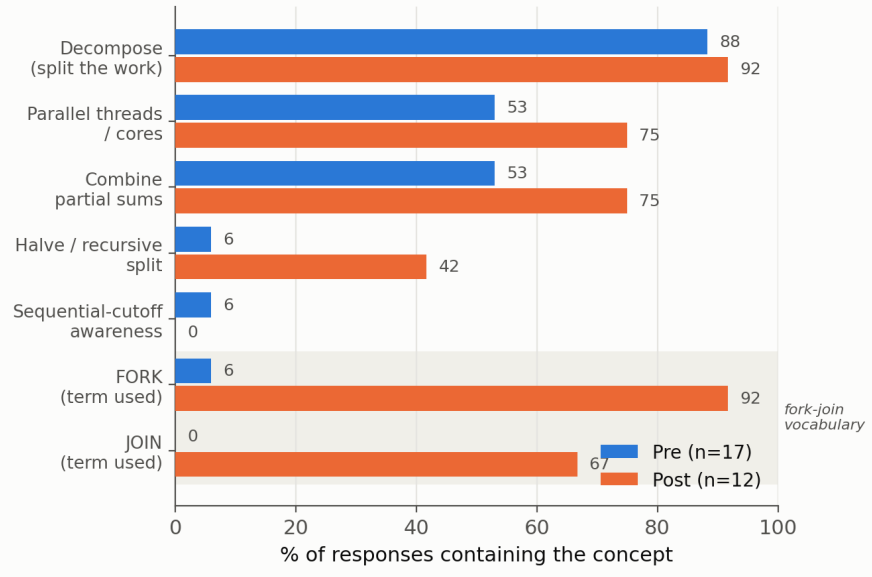


Figure 13.3: Pre/post analysis of the free-response answers.

13.4 New Plugged-in Activities

13.4.1 Greenfoot Plugged-In Activity

13.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

This activity was designed by researchers at Knox college and a more general description of the activity is available in Section 1.4.2. In this section, a version of the activity implemented in a CS2 class at USI is described. This plugged-in activity asks students, working in assigned teams, to design and build an original two-dimensional game in Greenfoot whose core mechanics are driven entirely by events. Greenfoot [34] is a Java-based educational IDE in which the world is populated by *actors* whose behavior is expressed through an `act()` method that the environment invokes repeatedly on every step of a simulation loop. Rather than writing an explicit main loop, students respond to events such as key presses, mouse clicks, timer ticks, and collisions between actors, and let the framework dispatch control to their handlers. The activity is intentionally open-ended: teams are given two reference projects, *Hedgehog* and *Zelda3* (see Figure 13.4 for a sample Greenfoot environment), demonstrating movement between screens, basic animation, simple enemies, a dialog window. Teams are explicitly required to produce a game that is meaningfully different from the provided examples, with original mechanics, assets, or gameplay features, rather than a minor modification of a sample.

This plugged-in activity is deliberately sequenced. It follows the introduction of the event-handling concept and the corresponding unplugged activity, in which each team drew and explained an "overly complicated machine" containing at least five distinct events (see Section 13.3.1). The unplugged activity builds the mental model of a system as a set of discrete event–response pairs. This plugged-in activity then asks students to realize that same model in working code. Prerequisites are therefore modest: prior programming ex-

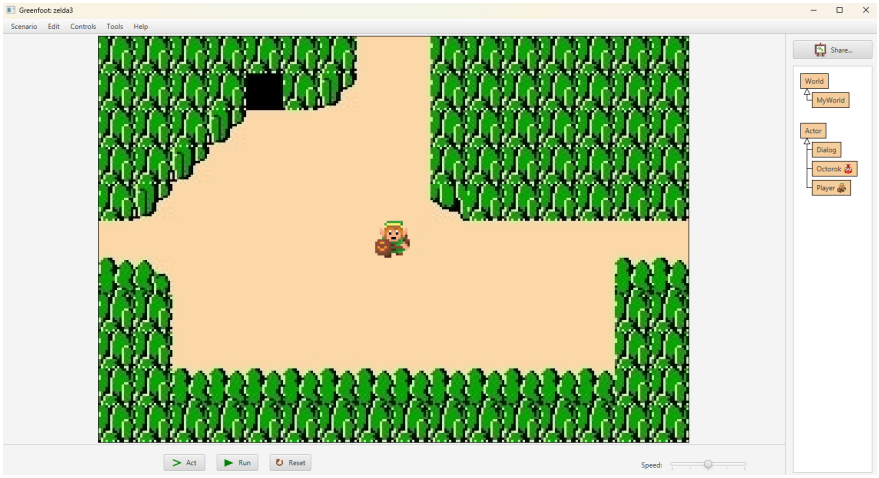


Figure 13.4: An example Greenfoot environment of the Zelda game.

perience with Java syntax, classes, objects, and methods, together with the conceptual grounding in events established by the preceding lecture and unplugged exercise. No prior game-development or Greenfoot experience is assumed.
By completing the activity, students should be able to:

1. identify the distinct events a program must respond to and map each to a handler;
2. implement keyboard-, mouse-, timer-, and collision-driven event handling across multiple interacting actors using Greenfoot's API (e.g., `Greenfoot.isKeyDown()`, `getOneIntersectingObject()`, and the per-step `act()` contract);
3. reason about a program whose control flow is dispatched by the framework rather than written as an explicit sequential loop;
4. integrate and manage external assets (custom sprites and images) through the project's `images` folder and the appropriate constructor code; and

13.4.1.2 Algorithms

The algorithmic content of the activity is the event loop and the dispatch of control to handlers, made concrete through Greenfoot's per-step execution model. The general shape each team implements is the following:

For each actor, on every step (`act`):
1. Read input / detect events (is a key held? was a key pressed this step? is the mouse clicked?).
2. For each detected event, run the corresponding handler (move, fire, change image, play sound).
3. Detect collisions with other actors (`getOneIntersectingObject`), and run the collision handler (score, remove actor, end round).
4. Advance any time- or state-based behavior (enemy advance, animation frame, restart timer).
5. Update the world and score.

13.4.1.3 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests).** The activity is supported by a collection of materials that can be found at this link: USI GitHub Greenfoot. A lecture introduces event-driven programming and the notion of events as discrete response points along with the unplugged activity described in Section 13.3.1. Two example projects, *Hedgehog* and *Zelda3*, are distributed as runnable Greenfoot projects and serve as the primary reference implementations. A short demo video produced by the creator of the *Zelda3* example accompanies the code and walks through its structure, including how custom images are stored and used. The written assignment specification defines the task, the submission process, and an explicit grading rubric assessing the event-driven implementation. In place of a conventional test, assessment is performance-based: the working game and a required team write-up describing three interesting or technically challenging features. The handout additionally embeds a short reflective survey (on the unplugged activity, the Greenfoot examples, and the project itself) used solely to evaluate the effectiveness of the assignment in teaching event-driven programming and not to affect grades.
- **How-To Guides** Include installation guidance points students to `https://www.greenfoot.org/download` and notes cross-platform support for Windows and macOS. A guide on using custom images instructs students to place image files in the `images` folder inside the Greenfoot project using their native file manager and to load them in an actor's constructor with a small provided code snippe. Also, direct students to `itch.io` or similar resources for artwork and downloadable asset packs, explains the term *assets* (buttons, icons, sprites, and similar artwork), and specify the professional requirement to credit any third-party assets and to keep a record of each asset's source link. Include submission instructions for uploading deliverables to the LMS (if applicable), and optionally collect a brief write-up (Word or PDF) covering three notable features and the reflective survey responses.

Table 13.9 provides information on duration effort required for implementing the activity in a CS2 class as a project.

13.4.1.4 Experience and Teaching Tips

The sequencing from unplugged to plugged-in worked well in practice: teams arrived at the Greenfoot task already thinking in terms of discrete events and handlers, which made the leap from the "overly complicated machine" diagram to code less complicated. Instructors should preserve this ordering rather than opening directly with the IDE. Providing two contrasting worked examples: one minimal (*Hedgehog*) and one feature-rich (*Zelda3*), is valuable because it gives weaker teams a base to follow while giving stronger teams a ceiling to reach past. Pairing the richer example with a creator's walkthrough video substantially reduced the number of setup and asset-loading questions. Two logistics tips help: keep teams small enough that everyone touches the event-handling code (rather than one member owning all the logic), and require the three-feature technical write-up, which pushes students to articulate why a given piece of event handling was non-trivial and consolidates the learning.

Table 13.9: Approximate Timing for the Greenfoot Plugged-In Activity Implemented as a Week-Long Project

Activity	Effort
Project kickoff: joining the assigned team, reading the specification, the grading rubric, and the reflective write-up questions	0.5 hr
Review of prerequisites: installing Greenfoot, revisiting the event-handling lecture and the unplugged "overly complicated machine" activity, and studying the `Hedgehog` and `Zelda3` example projects and the accompanying demo video	1 hr
Design: choosing a game concept "meaningfully different" from the examples and planning the actors and the events each must handle (keyboard, mouse, collision, timing)	1.5 hr
Asset preparation: sourcing or creating custom sprites (e.g., from itch.io), placing them in the `images` folder, and recording source links for citation	1 hr
Implementation: writing the actors and their `act()` event handlers, and integrating images via the `GreenfootImage` constructor	3 hr
Testing and debugging: play-testing mechanics and tuning event behavior (collision resolution, motion, spawning)	2 hr
Technical write-up: describing three interesting or technically challenging features and answering the reflective survey questions	1 hr
Subtotal (out-of-class effort)	**∼10 hr**
In-class support: worked-example walkthrough and instructor Q&A during the project window	remainder as needed
Total	**one week (team, primarily out of class)**

13.4.1.5 Data and Analysis

Assessment data were drawn from the written submissions of seven student teams (referred to here as Teams Alpha, Beta, Delta, Epsilon, Gamma, Kappa, and Pi). Each team's midterm submission included two components used as evidence: (i) a required technical write-up describing three interesting or technically challenging features the team implemented, and (ii) responses to a short reflective survey administered with the assignment. The survey posed three questions, framed so as not to affect the project grade and used solely to evaluate the effectiveness of the assignment in teaching event-handling:

1. How did you like the unplugged event-based activity done in class, where you and your team had to draw and explain an overly complicated machine with five events? Did it help you better understand the fundamentals of event-driven programming?
2. How do you like the event-driven examples in Greenfoot? Did they help you understand how event handling is implemented?
3. How did you like developing your own Greenfoot game using event-based programming, what was your favorite part, and do you have any suggestions for improvement?

Because the collected sample size is small and includes open-response answers, it was analyzed qualitatively rather than statistically. Two complementary analyses were applied. First, an attitudinal coding assigned each team's answer a sentiment (positive, mixed, or negative) together with a binary judgment of whether the team self-reported a learning benefit. Second, a learning-evidence coding cataloged the concrete event-handling constructs each team reported implementing in its game (keyboard input, mouse input, collision detection, timing/spawning, and custom sprite/asset integration), on the premise that the type and variety of events a team actually implemented is a more direct indicator of learning than

sentiment alone. Numeric totals from a seven-team sample are reported only to summarize the corpus and not to support any statistical inference.

Student attitudes toward all three components of the instructional sequence were uniformly positive. Table 13.10 summarizes the per-team sentiment. Six of the seven teams explicitly addressed the unplugged activity. Six teams rated it positively and reported that it aided their understanding of event-driven fundamentals. All seven teams rated the Greenfoot examples positively, and all seven rated the experience of building their own game positively.

Table 13.10: Per-team sentiment on the three reflective questions. + = positive; ∼ = mixed; ∘ = not addressed.

Team	Game (genre)	Q1: Unplugged	Q2: Examples	Q3: Own game
Alpha	Mouse-controlled shield	+	+	+
Beta	Platformer / physics jump	∘	+	+
Delta	Top-down alien shooter	+	+	+
Epsilon	Goose shooter (mouse aim)	+	+	+
Gamma	Timed sword game	+	+	+
Kappa	Platformer (spikes/prefabs)	+	+	+
Pi	Snake	+	+	+
Positive / addressed		6 / 6	7 / 7	7 / 7

Beyond self-reported attitude, the "three features" write-ups provide behavioral evidence that teams implemented genuine, and often non-trivial, event handling. Table 13.11 catalogs the event-handling constructs each team reported building. Collectively the cohort exercised the full range of event sources targeted by the activity and every team integrated custom sprites or assets through the `images` folder as the assignment intended. Several teams went beyond the class examples. For instance, Team Delta implemented enemy path-finding toward the player using the Pythagorean theorem for efficient diagonal movement and switched sprites according to movement and shooting direction; Team Epsilon built a mouse-cursor follower for aiming plus randomized enemy spawning; Team Beta replaced naive position changes with a momentum-based jump so the character "would stop only teleporting up"; and Team Alpha implemented cross-actor scoring logic that aggregated state from all actors and repurposed the mouse position as a collision hit box. These event-driven designs, consistent with the assignment's "meaningfully different" requirement, and the survey claim that the examples supported rather than substituted for the teams' own implementation work.

Table 13.11: Event-handling constructs reported per team (✓ ēxplicitly described in the write-up).

Team	Keyboard	Mouse	Collision	Timing/Spawn	Sprites/Assets
Alpha		✓	✓		✓
Beta	✓		✓		✓
Delta	✓		✓	✓	✓
Epsilon		✓	✓	✓	✓
Gamma	✓			✓	✓
Kappa	✓		✓	✓	✓
Pi	✓		✓	✓	✓

Every team's writeup reports that struggles are implementation-level (boundary cases, geometry, and tuning) rather than conceptual, which indicates that the preceding unplugged and worked-example scaffolding had established the underlying event-handling concept. The single most common suggestion, raised by four of the seven teams, was for more in-class collaboration time (Epsilon, "we should have had more time in-class to work on the project"; Team Beta and Gamma echoed wanting more time for polish and to explore Greenfoot further) and for more hands-on familiarity with the environment before the project. In future iterations, this can be addressed by a low-cost refinement with inserting a short, guided in-class Greenfoot lab between the worked examples and the open project.

13.4.2 Fork-Join Parallel Sum

13.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

After students had internalized the mechanics of divide-and-conquer parallelism through the unplugged post-it activity and the accompanying lectures on multithreading and the fork-join model, we asked them to carry the same ideas into working code. The plugged-in activity is delivered as Part A of the course's team-based final project. The problem is deliberately compute-bound so that the benefit of parallelism is observable on commodity hardware. Students must count all odd primes of the form $2i + 1$ for $1 \leq i < 999{,}999$. They are given two Java source files. The first, `SerialPrimes.java`, is a complete and correct serial reference implementation, where odd numbers are represented by their index i. The second, `ForkJoinPrimes.java`, is a skeleton containing the same `main` method that students must complete so that the identical count is produced in parallel using Java's Fork/Join framework.

The activity assumes the CS1/CS2 programming foundation that precedes it in the course. Students are expected to be comfortable with Java classes, methods, arrays, loops, and especially recursion, since the parallel solution is a recursive divide-and-conquer computation. They should understand the serial primality test well enough to reuse it as a base case, and they must be able to read and extend a class that uses inheritance and generics. Conceptually, the activity presumes the fork-join vocabulary developed in lecture and in the preceding unplugged exercise including the definitions of parallel speedup and Amdahl's law. By completing the activity, students should be able to:

- transform a serial divide-and-conquer computation into a parallel one using the Fork/Join framework;
- implement a `RecursiveTask` subclass with a correct base case governed by a threshold and a recursive case that splits the range, forks one half, computes the other, and joins;
- explain the role of the threshold in balancing available parallelism against task-management overhead;
- empirically measure elapsed time, compute speedup as $T_{\text{serial}}/T_{\text{parallel}}$, and interpret the result in light of the machine's core count and Amdahl's law; and
- articulate why observed speedup is sub-linear and how task granularity governs CPU utilization.

13.4.2.2 Algorithms

The first algorithm applied via this activity is the serial reduction that the students are given. The algorithm steps are:

1. iterate over every index i in $[1, 999{,}998]$,
2. test whether $2i + 1$ is prime by trial division up to $\sqrt{2i + 1}$, and
3. sum the number of primes found.

This is implemented as an ordinary sequential loop and serves as both the correctness oracle and the timing baseline.

The second is the parallel algorithm the students write, and it is the direct computational implementation of the unplugged summing activity. Counting primes in a range is a reduction, where the count over a range equals the count over its left half plus the count over its right half. The recursive task therefore follows the canonical fork-join shape. Given a sub-range of indices, the `compute` method first checks whether the range is smaller than the threshold; if so, it has reached the base case and counts primes directly with the serial routine. Otherwise it splits the range at its midpoint, constructs a left and a right sub-task, forks the left sub-task so it may run on another worker, computes the right sub-task in the current thread, then joins the left sub-task to obtain its result and returns the sum of the two partial counts. Recursion continues to split the range until sub-ranges fall below the threshold, producing exactly the binary tree of tasks the students had already drawn by hand during the unplugged activity.

13.4.2.3 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** The activity is scaffolded by a dedicated lecture module on multithreading and fork-join parallelism. The material can be found online at USI GitHub Plugged-In Fork-Join. Using a running "sum an array" example, the lectures develop the conceptual `fork/join` keywords, generalize from fixed thread counts to a recursive formulation with a threshold, and finally show the concrete Java implementation as a `RecursiveTask` in (`sumArray.java`), framing the pattern as a general reduction. Immediately preceding all of this is the unplugged post-it activity, in which a pool of fifteen students sums the values 1 to 32 with a cutoff of four, physically dividing ranges, joining partial sums, and passing results up the tree. The plugged-in assignment itself is the graded deliverable, where students complete `ForkJoinPrimes.java`, answer the embedded discussion questions in a header comment, and present their threshold-experiment results in class as part of the final project.
- **How-To Guides** The assignment provides step-by-step guidance that mirrors the algorithmic decomposition. Students are directed first to read `SerialPrimes.java` carefully and to note the $2i + 1$ indexing and the `isPrime` primality test. They are then walked through completing the parallel task with a threshold-governed base case, create a single task over the full range $[1, 999{,}999)$, invoke it through `ForkJoinPool.commonPool().invoke(...)`, and print the returned count. Students vary the threshold across a supplied set of values (500,000; 250,000; 100,000; 10,000; 1,000; 100; and a serial baseline), run each configuration at least three times,

and record the runs, their average elapsed time, an estimate of the number of tasks created, and the resulting speedup in a provided table. The students answer the following reflective questions: how elapsed time changes as the threshold decreases and whether a sweet spot exists; what overhead is observed when the threshold is very small (e.g. 100); and how the best parallel time compares to the serial time as a speedup ratio.

Table 13.12 provides information on duration effort required for implementing the activity in a CS2 class as a project.

Table 13.12: Approximate Timing for the Fork-Join Plugged-In Activity Implemented Part A of a Week Long Project

Activity	**Effort**
Project kickoff: team formation, reading the specification and the discussion questions	0.5 hr
Review of prerequisites: revisiting the fork-join lecture and the unplugged Post-its activity, and studying `SerialPrimes.java`	1 hr
Design: planning the parallel solution with a threshold-governed base case and a recursive split–fork–compute–join structure	1 hr
Implementation: completing `ForkJoinPrimes.java`	1.5 hr
Testing and debugging	1.5 hr
Threshold experiment: running each of the six thresholds and the serial baseline at least three times, recording elapsed times, and computing the average time and speedup in the results table	1.5 hr
Discussion write-up: answering the reflection questions in the file header comment	0.5 hr
Presentation preparation: assembling the threshold table and findings for the in-class final-project presentation	1 hr
Subtotal (Part A)	**∼8.5 hr**
Part B: second project component, completed concurrently by the team	remainder of week
Total	**one week (team, out of class)**

13.4.2.4 Experience and Teaching Tips

The strongest pedagogical lever we observed was the three-stage scaffolding: unplugged embodiment, then lecture formalization, then a plugged-in activity. By the time students opened `ForkJoinPrimes.java` the shape of the solution was already familiar. We recommend preserving that ordering rather than leading with the framework API. Choosing prime counting rather than array summation for the graded task was deliberate because summation is so cheap that thread overhead swamps any speedup and can leave students concluding, incorrectly, that parallelism does not help, whereas the compute-bound primality test produces a clear, measurable speedup that makes the theory tangible.

Several recurring pitfalls are worth anticipating. Students frequently retain the serial version's shared mutable counter. It is worth stressing that the parallel design must return partial results up the tree and combine them, not change shared state, and the assignment explicitly asks that this field be removed. Off-by-one errors at the midpoint split, and forgetting to `join` before using a forked result, are common and map neatly back to the precedence-constraint discussion from lecture. On the experimental side, students are often surprised that speedup is sub-linear and caps below the number of cores; this is the moment to connect their table to Amdahl's law and to the overhead slide, and to note that the common

pool's default parallelism is tied to the available processors. Finally, requiring multiple runs and reporting the average (as the worksheet does) tempers spurious conclusions and models sound benchmarking practice.

13.4.2.5 Data and Analysis

In Part A of the project, student teams implemented a Java Fork/Join program (counting primes over a large numeric range), swept the sequential threshold across several orders of magnitude, timed each configuration against a serial baseline, and then answered three short-response questions interpreting their own measurements:

Q1. How does the elapsed time change as the threshold decreases? Is there a sweet spot, and why?

Q2. What happens when the threshold is very small (e.g., 100)? What overhead appears?

Q3. How does the best parallel time compare to the serial time? What is the speedup?

This section analyzes the written responses of the seven teams ($N = 7$) that completed the activity in the Spring 2026 offering. Because the responses are open-ended team artifacts rather than scored items, we use a qualitative thematic analysis. We first tabulate the quantitative claims each team reported (Table 13.13), then code every response against a small set of target concepts drawn directly from the activity's learning outcomes (Table 13.14), and finally read the responses closely for the reasoning and the residual misconceptions behind the codes.

Reported experimental results. Table 13.13 collects what each team reported. Every team that supplied raw timings observed a genuine speedup (> 1), and, where a team reported both its raw times and a computed ratio, we re-derived the speedup as $T_{\text{serial}}/T_{\text{parallel}}$ to check the arithmetic (final column).

Table 13.13: Quantitative claims reported by each team. "Sweet-spot threshold" is the cutoff each team identified as best. The verified speedup re-computes $T_{\text{serial}}/T_{\text{parallel}}$ from the team's own timings.

Team	Sweet-spot threshold	Serial (ms)	Parallel (ms)	Reported speedup	Verified speedup
AbstractAces	1,000	280.66	125.33	2.23	2.24
ByteCoders	1,000	1180	23	51.3	51.3
MethodActors	100,000	117	22.33	5.24	5.24
Polymorphs	(trend only)	685.6	385.3	1.78	1.78
RecursiveRebels	250,000	738	409	1.80	1.80
SyncSquad	1,000	224	21	11.28	10.67
ThreadRunners	1,000	91	34	2.68	2.68

Thematic coding. We coded each team's three-part response against six concepts that the activity is designed to develop. Each concept is marked present (yes), partially present (pt), or absent (no) for each team in Table 13.14.

Table 13.14: Concept coverage across teams. yes for present, pt for partial, no for absent. G1–G6 are defined in the text.

Team	G1 granularity	G2 threshold	G3 overhead	G4 task count	G5 speedup	G6 h/w
AbstractAces	yes	yes	yes	no	pt	pt
ByteCoders	yes	yes	yes	no	yes	pt
MethodActors	yes	yes	no	no	pt	yes
Polymorphs	yes	yes	yes	no	yes	pt
RecursiveRebels	yes	yes	yes	yes	yes	pt
SyncSquad	yes	yes	yes	no	pt	pt
ThreadRunners	yes	yes	yes	yes	yes	pt
Present (yes)	7/7	7/7	6/7	2/7	4/7	1/7

The six concepts are: **G1 (granularity control)**: decreasing the threshold produces more, smaller tasks and thus finer-grained parallelism; **G2 (the sweet-spot trade-off)**: performance improves as tasks are added but eventually degrades, so an optimum exists; **G3 (parallel overhead)**: naming task creation, scheduling, forking, and joining as the cost that eventually dominates; **G4 (quantified task explosion)**: giving a concrete count of the tasks generated at a small threshold; **G5 (speedup computed)**: applying $T_{\text{serial}}/T_{\text{parallel}}$ to real timings; and **G6 (hardware awareness)**: recognizing that a finite number of cores bounds the achievable speedup.

Read as a whole, the team responses demonstrate that the plugged-in activity consolidated the concepts introduced physically in the unplugged one. Every team reasoned correctly about task granularity and the existence of an optimal cutoff. A large majority named the specific fork/join overhead responsible for the degradation and most could compute and interpret speedup from their own measurements. The strongest responses fused vocabulary, mechanism, and measured quantities in a single explanation, while the weaker ones erred by conflating overhead with core scarcity or by a minor arithmetic slip. None of the teams missed the parallel-decomposition model understanding itself.

13.4.3 Seeing Concurrency: A Java-Based Parallel Image Processing

13.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

This activity introduces the concept of data parallelism, multithreading, and parallel computing. The main activity was developed by educators at Montclair State University and a description of the original activity can be found in Chapter 10, Section 10.4.4.

The CS2 students at USI are given a fully working sequential Java program that performs Sobel edge detection on an image, and are asked to redesign the work distribution so that multiple threads compute the output concurrently, reducing wall-clock runtime on multicore hardware. The program applies the Sobel operator to every interior pixel of an image. Because each output pixel depends only on the 3×3 neighborhood of a read-only input image, all pixels can be computed independently with no data races. This makes the problem an

"embarrassingly parallel" workload and an ideal first exposure to shared-memory parallelism.

The activity assumes students can read and write intermediate Java, including classes, methods, arrays, and nested loops, and are comfortable compiling and running a small Java project from the command line or an IDE such as VS Code. No prior experience with threads is required. The necessary concurrency constructs are introduced in the accompanying background material. A conceptual familiarity with images as two-dimensional pixel grids is helpful but not essential.

Upon completing the activity, students should be able to: (1) recognize when a computation is data-parallel and free of data dependencies; (2) decompose a nested-loop workload into independent tasks and distribute them across a pool of worker threads; (3) use Java's `ExecutorService` API: `Executors.newFixedThreadPool`, `submit`, and `shutdown` to manage a thread pool; (4) coordinate thread completion with a `CountDownLatch` so that timing and output occur only after all work finishes; and (5) measure, tabulate, and reason about parallel speedup, relating their observations to Amdahl's Law and to concrete sources of parallel overhead.

13.4.3.2 Algorithms

The core algorithm is Sobel edge detection. However, students do not need prior knowledge of the algorithm or a deep understanding of its implementation to apply parallelism. For each interior pixel (x, y), the grayscale image is convolved with two 3×3 kernels that estimate the horizontal and vertical intensity gradients:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}.$$

Each RGB pixel in the neighborhood is first reduced to a grayscale intensity by averaging its red, green, and blue channels. The two weighted sums sum_X and sum_Y are combined into a gradient magnitude, clamped to the valid intensity range: $\text{magnitude} = \min\bigl(255, \sqrt{\text{sum}_X^2 + \text{sum}_Y^2}\bigr)$. The magnitude is packed into a grayscale RGB integer and written to the corresponding pixel of the output image. Border pixels are skipped so that every kernel access stays in bounds, which is why the loops run from index 1 to height $-$ 2 and 1 to width $-$ 2.

The parallelization algorithm is a row-level task decomposition. Rather than the single main thread sweeping every row, the outer loop over rows is turned into a set of independent tasks: each row y (for y from 1 to height $-$ 2) becomes one `Runnable` that computes every pixel in that row and is submitted to a fixed-size thread pool. The pool's worker threads pull tasks from an internal queue and execute them concurrently across the available cores. Correctness is preserved because tasks only read the shared input image and write to disjoint pixels of the output image, so no synchronization on the pixel data is needed. Completion is coordinated with a `CountDownLatch` initialized to the number of row tasks (height $-$ 2): each task calls `countDown()` when its row is done, and the main thread calls `await()` before stopping the timer, shutting down the executor, and writing the output file. This guarantees the reported time reflects the full parallel computation.

13.4.3.3 Implementation Details

The instructional material used for implementing this activity can be found at the following link: USI GitHub Parallel Image Processing.

- **Instructional Material (Lectures, Videos, Assignments, Tests)** The activity is delivered as the second part (Part B) of a final project. A short background section motivates the two building blocks students will need. The first is the Sobel edge-detection algorithm itself, presented with its two convolution kernels and the magnitude formula, alongside the key observation that the read-only input and neighborhood-local output make the loop safe to parallelize. The second is a focused introduction to Java's `ExecutorService`, limited to exactly the API surface the assignment uses: `Executors.newFixedThreadPool(n)` to create a pool of n workers, `executor.submit(Runnable)` to enqueue a unit of work, `executor.shutdown()` to signal that submission is complete, and `CountDownLatch` for barrier-style synchronization between the workers and the main thread. The provided sequential program, `EdgeDetectionSeq.java`, doubles as a lecture artifact and a specification. Students study it to understand the serial control flow and the two helper methods (`computeEdgePixel` and `rgbToGray`) they will reuse unchanged. Students implement `EdgeDetectionParallel.java` to satisfy an explicit checklist: a configurable thread count; row-level work distribution matching the sequential loop bounds; correct `CountDownLatch` synchronization with the timing block enclosing the `await()` call; verbatim reuse of the two helper methods; and output written to `images/output/image_edges_parallel.jpg` using the same "Execution time: X ms" format as the serial version. The assessment component is a performance-analysis task that functions as an embedded test of conceptual understanding: students run their code at 1, 2, 4, 8, and 16 threads (two runs each, averaged), tabulate execution time and speedup ($T_{\text{sequential}}/T_{\text{parallel}}$), and answer three short-response questions in a header comment covering scalability and Amdahl's Law, the optimal thread count for their machine, and at least two concrete sources of parallel overhead. An in-class presentation of the implementation and the results table completes the activity.
- **How-To Guides** Step-by-step guidance walks students from a working serial program to a measured parallel one. First, they unzip the starter project, which contains `EdgeDetectionSeq.java`, a place for `EdgeDetectionParallel.java`, and `images/input` (see Figure 13.5 for a sample input image) and `images/output` folders with one sample image provided; they compile and run the sequential version (`javac EdgeDetectionSeq.java` then `java EdgeDetectionSeq`, or the equivalent in VS Code) to confirm correct output before changing anything. Next, they create `EdgeDetectionParallel.java` following the requirement checklist above, using the sequential file as the reference for structure and reusing the helper methods without modification. Finally, the guide directs the performance study by fixing a single input image across all runs, varying only the thread count, averaging two runs per configuration, and filling in the results table, so that measurements are comparable and the speedup analysis is meaningful. Table 13.15 lists the approximate timing for the various stages of implementation of the activity in a CS2 class.



Figure 13.5: An example input image used for the parallel image processing activity.

Table 13.15: Approximate Timing for the Parallel Image Processing Plugged-In Activity

Activity	**Effort**
Project kickoff (in class): team formation, reading the specification and the discussion questions	15 mins
Review of prerequisites (in class): revisiting the data-parallelism and multithreading lecture, the `ExecutorService` and `CountDownLatch` background, and studying `EdgeDetectionSeq.java`	15 mins
Design (offline): planning the parallel solution with a configurable thread pool and a row-level split–submit–count-down–await structure	1 hr
Implementation (offline): completing EdgeDetectionParallel.java	1.5 hr
Testing and debugging (offline)	1.5 hr
Thread-count experiment (offline): running the serial baseline and each of the four thread counts (2, 4, 8, 16) at least twice, recording elapsed times, and computing the average time and speedup in the results table	1.5 hr
Discussion write-up (offline): answering the reflection questions in the file header comment	0.5 hr
Presentation preparation (offline): assembling the thread-count table and findings for the in-class final-project presentation	1 hr
Subtotal (Part B)	**∼7.5 hr**
Part A: first project component, completed concurrently by the team	remainder of week
Total	**one week (team, out of class)**

13.4.3.4 Experience and Teaching Tips

Row-level decomposition is the sweet spot for a first parallel program as it maps cleanly onto the existing outer loop, gives each task enough work to matter, and avoids the false-sharing and fine-grained-overhead pitfalls that per-pixel tasks would introduce. Instructors should expect students to reach for per-pixel tasks first. Steering them toward one task per row is a good discussion of task granularity. Emphasize early why this workload is race-free because that reasoning, not the API mechanics, is the transferable PDC lesson.

The most common correctness bug is a timing block that does not enclose `latch.await()`, which makes the parallel version look impossibly fast because the timer stops before the workers finish. Requiring the latch and the enclosing timer up front heads this off. Encourage students to verify that the parallel output image is pixel-identical to the sequential one before trusting any speed numbers.

Instructors are encouraged to set clear expectations for the performance table. Speedup will not scale linearly, and students should observe diminishing returns once the thread count exceeds the machine's physical core count, which motivates Amdahl's Law and the notions of serial fraction. Because results depend heavily on core count, image size, and JVM warm-up, results vary across student laptops. Framing this variability as data to explain leads to the strongest classroom discussions and helps students identify genuine sources of overhead. If time allows, a larger input image sharpens the speedup curve by increasing the compute-to-overhead ratio and makes the multi-core benefit more visible.

13.4.3.5 Data and Analysis

Seven student teams submitted the reflection questions and a timing table for the parallel image edge-detection activity. Because each team ran on its own laptop, absolute execution times are not comparable across teams. Sequential baselines ranged from roughly 589 ms to over 2000 ms, so the analysis below relies on speedup ($T_{\text{sequential}}/T_{\text{parallel}}$), which normalizes each team to its own single-threaded baseline. Table 13.16 consolidates the reported results.

Table 13.16: Consolidated Per-Team Results for the Parallel Image Processing Activity

		Speedup at thread count				
Team	**Peak**	**Seq**	**2**	**4**	**8**	**16**
AbstractAces	8	1.00	1.55	2.51	**2.66**	2.50
ByteCoders	16	1.00	1.96	2.97	3.31	**3.64**
MethodActors	16[†]	1.00	1.44	2.23	2.78	**2.88**
Polymorphs	16	1.00	1.79	2.34	3.34	**3.59**
RecursiveRebels	16	1.00	1.57	2.35	2.72	**2.79**
SyncSquad	8	1.00	1.78	3.21	**4.30**	3.89
ThreadRunners	16	1.00	1.30	2.07	3.16	**3.40**

[†]MethodActors reported 8 as optimal; their 16-thread time was only marginally faster (a plateau, not a gain).

Speedup magnitude and sublinearity. No team achieved linear speedup, exactly as the activity intends. At two threads the teams clustered between 1.30× and 1.96×, so even

the best-case doubling of resources fell short of a 2× return. The gap widened with scale. At eight threads, where an ideal machine would show 8×, measured speedups ranged only from 2.66× to 4.30×. The peak speedup observed across all teams was SyncSquad's 4.30× at eight threads. This pervasive sublinearity is the central empirical result and the concrete evidence for Amdahl's Law, which states that the serial fraction of the program is not accelerated by additional threads and caps the attainable speedup well below the thread count.

Optimal thread count. Five of the seven teams recorded their fastest time at sixteen threads, while AbstractAces and SyncSquad both peaked at eight and then regressed at sixteen (AbstractAces 2.66× → 2.50×; SyncSquad 4.30× → 3.89×). Even among the teams whose sixteen-thread run was nominally fastest, the eight-to-sixteen improvement was small. MethodActors (2.78× → 2.88×) and RecursiveRebels (2.72× → 2.79×) show essentially a plateau, whereas the four-to-eight step still delivered a clear gain for every team. The data therefore tell a consistent story of diminishing returns setting in around the physical core count, with two teams crossing into the regime where extra threads cost more than they return. This matches the teams' stated reasoning, where several correctly attributed the plateau or regression to a limited number of cores or logical processors (SyncSquad cited "8 logical processors"; MethodActors and AbstractAces referenced the core count directly, AbstractAces noting that threads began to "compete for CPU time").

Quality of student reasoning. The three reflection questions probed distinct concepts, and the written answers varied in depth. On *scalability*, most teams correctly identified the speedup as sublinear and tied it to Amdahl's Law. RecursiveRebels gave the most complete answer, enumerating the specific non-parallel stages (image loading, task creation, latch waiting, image writing, and scheduling). Polymorphs was the notable outlier, asserting that their data did scale linearly as a misreading, since their own table shows only 3.59× at sixteen threads. On *optimal thread count*, the strongest answers connected the plateau to hardware parallelism and named oversubscription, context switching, and cache pressure as the mechanisms that diminish further gains. On *overhead*, the best responses (AbstractAces, MethodActors, SyncSquad, RecursiveRebels, ThreadRunners) correctly identified the two implementation specific sources the assignment targeted: task-management overhead from submitting one `Runnable` per image row (hundreds of small tasks through the `ExecutorService`) and synchronization overhead from the per-row `countDown()` and the main thread's `await()`. Polymorphs again fell short, citing generic hardware and image-quality factors rather than anything specific to the parallel implementation.

Data-integrity observations. Two submissions had issues that can be considered as instruction related issues. ByteCoders submitted a complete, well-formed timing table but left all three reflection questions blank. Several teams also mishandled the sequential row's speedup entry, leaving it blank (ByteCoders) or marking it "N/A" (ThreadRunners) rather than the definitional 1.00. These artifacts suggest a recurring recommendation for future offerings to require teams to report speedup relative to their own sequential baseline and confirm that every reflection question is answered before submission.

13.5 Course-wide Pre and Post Course Surveys

To evaluate the impact of integrating PDC concepts into the intermediate computer science curriculum, a comparative data collection study was conducted across two consecutive Spring semesters in the CS2 course. Baseline data was collected in Spring 2025 for a traditional CS2 curriculum structure focused strictly on sequential object-oriented programming without any explicit PDC interventions. To assess student growth and conceptual shifts, identical pre- and post-test surveys were administered during the first and final weeks of the semester. The same pre- and post-survey methodology was employed in Spring 2026 to measure the efficacy of a targeted pedagogical intervention. This intervention was structurally delivered through the two unplugged exercises and the two plugged-in coding modules discussed above.

By maintaining Java as the instructional language across both cohorts, the study isolated the curriculum redesign as the primary variable. The comparative results strongly indicate that the active-learning PDC intervention effectively improved CS2 students' self-reported understanding and confidence in PDC concepts.

13.5.1 Results and Analysis : Spring 2025 Survey Data

Pre- and post-course surveys were linked by student email address. Twenty students completed the pre-survey and twelve completed the post-survey. Every post-survey respondent was matched to a pre-survey record, yielding 12 complete pre/post pairs. Because this cohort received no parallel-and-distributed-computing (PDC) intervention, it serves as a baseline against which future intervention cohorts can be compared. Students rated their experience with 24 programming and data-structure topics and their agreement with 5 attitude statements, all on 7-point scales, and answered one open-ended question about their learning goals. The matched sample comprised 11 Computer Science majors and 1 CS/Math double major; 6 first-year students, 3 sophomores, and 3 juniors. Only items appearing on both surveys with identical question codes were analyzed. Demographic questions and the letter-grade expectation item (a different scale) were excluded. This left 29 comparable Likert-type items: 16 data-structure experience items (E1), 4 programming-practice items (E2), 4 advanced/PDC concept items (E3), and 5 attitude items (Q1). Experience responses were coded 1–7 (No experience at all = 1 . . . Extremely experienced = 7) and agreement responses 1–7 (Strongly disagree = 1 . . . Strongly agree = 7). Each item was tested with the Wilcoxon signed-rank test on matched pairs, appropriate for ordinal Likert data. Effect sizes are reported as Cliff's delta. Averaging items into construct composites gives the clearest summary. All three experience composites rose significantly with large effects, while neither PDC-focused composite reached significance. These results are summarized in Table 13.17.

Figure 13.6 plots every one of the 29 items grouped by the cocept construct. The bars represent the size of the shift from the pre mean to the post mean. The figure illustrates how the significant gains concentrate in data structures and programming practices, while the advanced/PDC and attitude blocks contain most of the small, non-significant movements.

Both surveys asked the identical open question: "What specific skills or abilities do you aspire to acquire or improve upon as a result of your learning in this class?" Therefore, comparing responses shows how students' stated goals evolved across the semester. All 12

Table 13.17: USI CS2 Spring 2025 Pre/Post Survey Results, Baseline (No PDC Intervention)

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Data structures (E1)									
classes&objects	12	5.50	6.08	0.58	3	0.094	No	−0.35	medium
events	12	4.17	5.33	1.17	0	0.004	Yes	−0.42	medium
lists	12	5.17	5.83	0.67	9	0.117	No	−0.29	small
generics	12	3.08	4.92	1.83	2	0.008	Yes	−0.59	large
stacks	12	2.33	5.00	2.67	0	0.004	Yes	−0.71	large
queues	12	2.58	5.00	2.42	5.5	0.021	Yes	−0.67	large
sorting	12	2.83	3.92	1.08	15	0.070	No	−0.40	medium
comparators	12	2.83	3.75	0.92	17.5	0.100	No	−0.24	small
recursion	12	3.00	4.67	1.67	6	0.016	Yes	−0.53	large
maps	12	2.42	3.00	0.58	14	0.320	No	−0.21	small
sets	12	2.92	4.50	1.58	3.5	0.023	Yes	−0.53	large
trees	12	2.75	3.42	0.67	18	0.369	No	−0.18	small
binarySearchTrees	12	2.33	3.00	0.67	10.5	0.375	No	−0.22	small
priorityQueues	12	2.00	3.17	1.17	12.5	0.125	No	−0.33	medium
hashing	12	2.00	2.92	0.92	2	0.031	Yes	−0.26	small
graphs	12	2.08	3.33	1.25	4	0.027	Yes	−0.42	medium
Programming Practices (E2)									
Writing test cases	12	3.58	5.08	1.50	3	0.006	Yes	−0.65	large
Event handling	12	4.58	5.83	1.25	0	0.002	Yes	−0.64	large
Choosing the correct data structure	12	4.08	5.58	1.50	9	0.032	Yes	−0.73	large
Writing classes	12	5.25	6.08	0.83	3.5	0.027	Yes	−0.46	medium
PDC Concepts (E3)									
Problem decomposition	12	5.58	6.17	0.58	15.5	0.146	No	−0.38	medium
Parallel processing	12	3.17	4.00	0.83	5	0.156	No	−0.28	small
Distributed computing	12	2.42	3.25	0.83	8.5	0.125	No	−0.31	small
Event handling	12	5.00	6.08	1.08	4	0.008	Yes	−0.50	large
Computing Attitudes (Q1)									
computing interest	12	6.75	6.75	0.00	10.5	1.000	No	0.00	negligible
PDC interest	12	6.17	6.17	0.00	7.5	1.000	No	0.01	negligible
PDC importance	12	6.25	6.58	0.33	3.5	0.219	No	−0.28	small
computing understanding	12	5.42	6.08	0.67	2.5	0.156	No	−0.22	small
PDC understanding	12	4.08	4.83	0.75	9	0.074	No	−0.43	medium

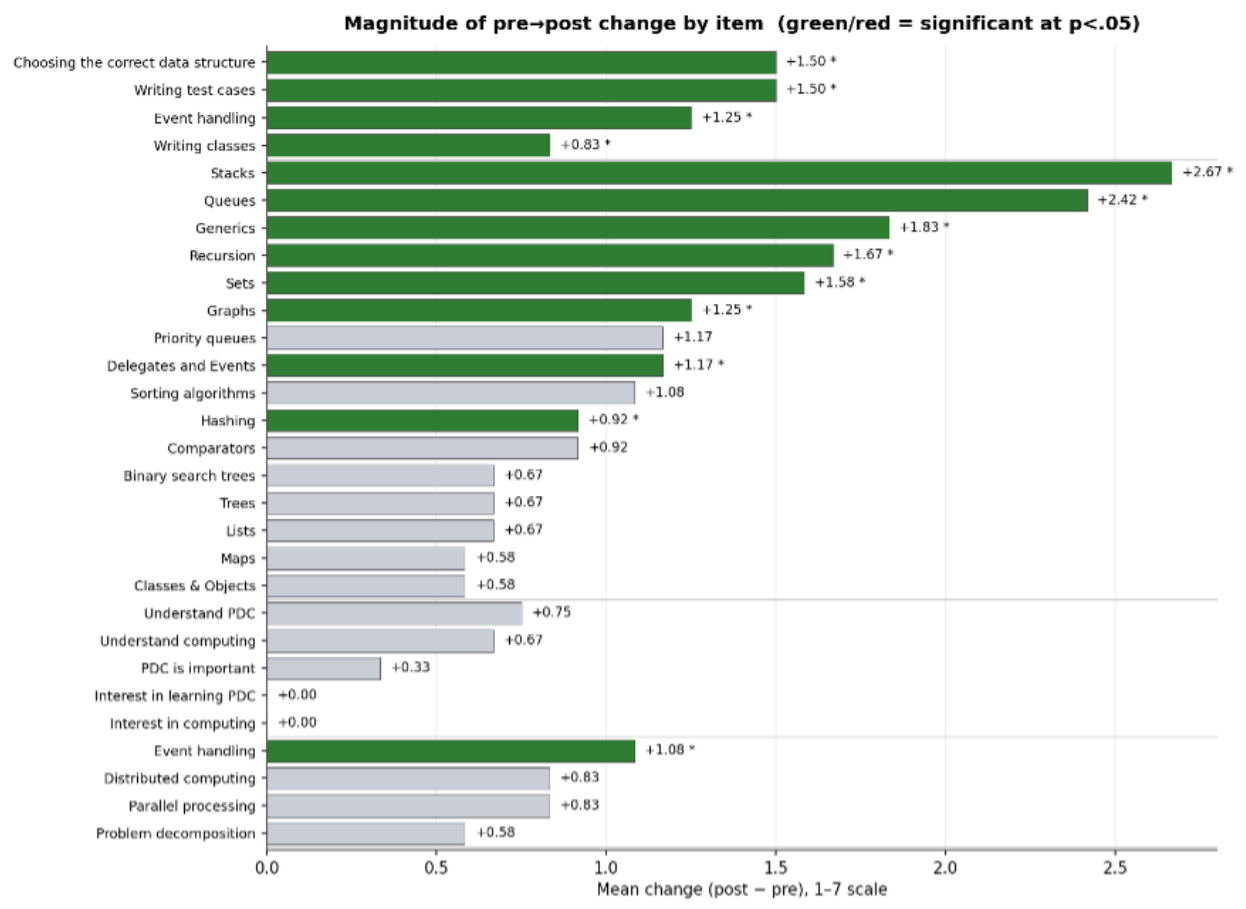


Figure 13.6: Pre to Post shift for every survey item (n = 12), grouped by construct. Green = significant increase ($p < .05$); grey = non-significant.

matched students answered at both points. Responses were read and coded thematically and the theme counts is shown in Figure 13.7.

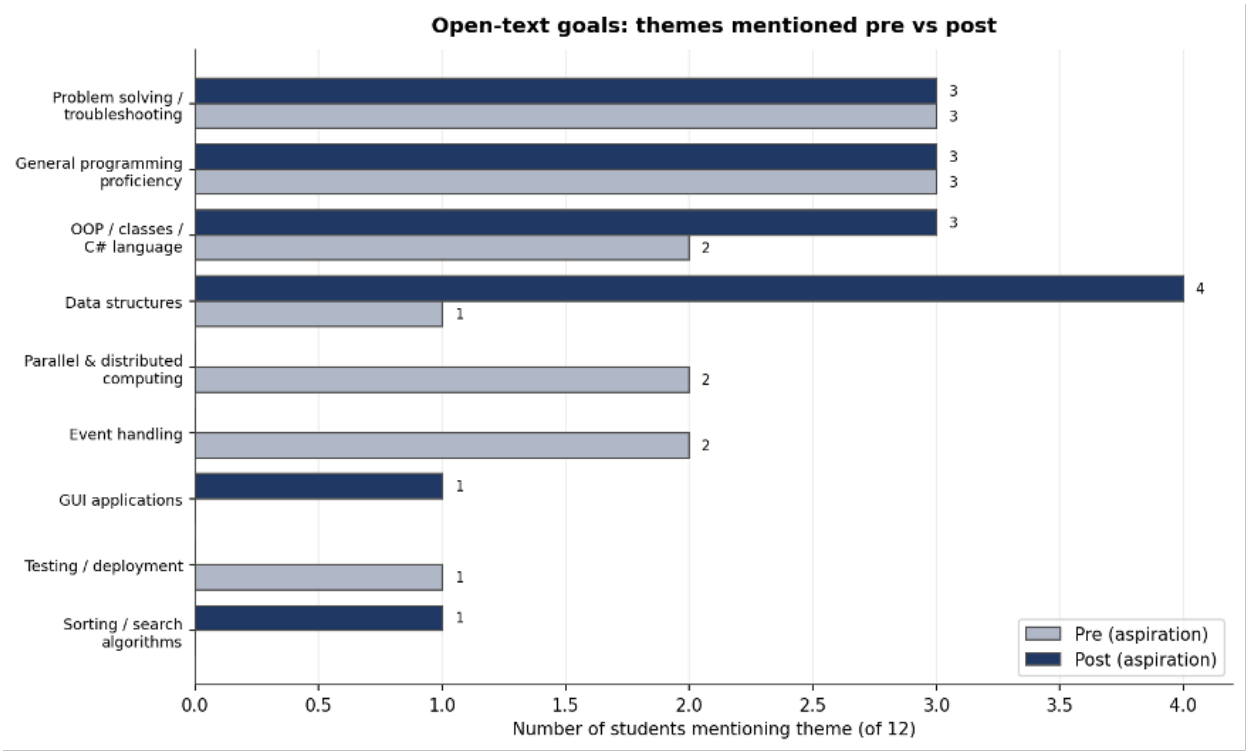


Figure 13.7: Themes in open-text learning goals, pre vs post (a student may express more than one theme)

13.5.2 Results and Analysis : Spring 2026 Survey Data (PDC Intervention)

The analysis reported here uses matched pre- and post-course experience surveys from the Spring 2026 offering of the CS2 course at USI. During this semester, the PDC intervention was delivered. Of all students surveyed, 18 completed both the pre- and post-course instruments and were matched on their email. Only these 18 matched records are included, and students who took only one survey were filtered out. The comparable items fall into four construct groups: data structures & language constructs (16 items), programming skills (4), PDC & problem-decomposition concepts (4), and attitudes & self-efficacy (5). Experience items were coded on a 7-point scale (1 = "No experience at all" ... 7 = "Extremely experienced"), agreement items on a 7-point Likert scale (1 = "Strongly disagree" ... 7 = "Strongly agree"). The open-ended aspiration item (Q2) was analyzed qualitatively, and the expected-grade item was tabulated separately because it uses a letter-grade scale. Because Likert data are ordinal and the design is paired, each item was tested with the Wilcoxon signed-rank test. Practical significance was estimated with Cliff's delta. These results are summarized in Table 13.18.

Table 13.18: USI CS358 (CS2) Pre/Post Survey Results, Spring 2026

Category	n	Mean (pre)	Mean (post)	Mean Diff	W	p-value	Sig?	Cliff δ	Interpretation
Data structures (E1)									
classObject	18	4.56	5.28	0.72	2.5	0.016	Yes	−0.33	medium
events	18	2.67	4.72	2.06	3	0.001	Yes	−0.68	large
lists	18	3.78	4.83	1.06	9	0.016	Yes	−0.36	medium
generics	18	2.33	3.94	1.61	2.5	0.002	Yes	−0.56	large
stacks	18	2.22	3.67	1.44	3	0.003	Yes	−0.50	large
queues	18	2.72	3.67	0.94	14	0.046	Yes	−0.30	small
sorting	18	4.00	5.00	1.00	0	0.001	Yes	−0.41	medium
comparators	18	3.50	4.67	1.17	11.5	0.030	Yes	−0.32	small
recursion	18	2.89	4.78	1.89	4.5	0.001	Yes	−0.55	large
maps	18	2.50	3.39	0.89	7	0.019	Yes	−0.34	medium
sets	18	3.17	3.83	0.67	14.5	0.094	No	−0.28	small
trees	18	2.89	4.33	1.44	3.5	0.002	Yes	−0.51	large
binarySearchTrees	18	3.56	4.33	0.78	27	0.056	No	−0.24	small
priorityQueues	18	2.50	3.67	1.17	5	0.002	Yes	−0.41	medium
hashing	18	3.06	3.56	0.50	35	0.261	No	−0.25	small
graphs	18	3.39	3.94	0.56	30	0.148	No	−0.23	small
Programming practices (E2)									
E_Writing test cases	18	3.11	4.78	1.67	2.5	0.002	Yes	−0.54	large
E_Event handling	18	2.89	5.17	2.28	0	0.001	Yes	−0.70	large
E_Choosing the correct data structure	18	3.56	5.06	1.50	7	0.002	Yes	−0.49	large
E_Writing classes	18	4.39	5.56	1.17	0	0.001	Yes	−0.49	large
PDC concepts (E3)									
Problem decomposition	18	4.44	5.22	0.78	6	0.014	Yes	−0.36	medium
Parallel processing	18	3.94	4.89	0.94	22.5	0.016	Yes	−0.38	medium
Distributed computing	18	3.44	4.28	0.83	41	0.087	No	−0.29	small
Event handling	18	3.50	5.33	1.83	0	0.003	Yes	−0.60	large
Computing Attitudes (Q1)									
interest in computing is high	18	5.89	5.89	0.00	18	1.000	No	0.03	negligible
interest in learning about PDC is high	18	5.50	5.33	−0.17	22.5	0.589	No	0.09	negligible
learning about PDC is important	18	5.83	6.06	0.22	32.5	0.317	No	−0.12	negligible
good understanding of computing	18	5.06	5.44	0.39	3	0.053	No	−0.19	small
good understanding of PDC	18	4.72	5.28	0.56	16.5	0.059	No	−0.22	small

Figure 13.8 shows the pre to post shift for all 29 items, sorted by the size of the gain. Students rated themselves more experienced or more confident on every construct after the course, with only the two "interest" attitude items essentially flat. The largest gains were on Event handling as a skill ($\Delta = +2.28$), Events ($+2.06$), Recursion ($+1.89$), Event handling as a concept ($+1.83$), and Writing test cases ($+1.67$). These reflect PDC topics with substantial hands-on coverage during the term.

The open-ended item asked students what specific skills or abilities they aspired to acquire or improve. Responses from all 18 matched students were coded thematically at both time points. The most striking finding is a shift from general, non-specific goals before the course toward concrete, course-aligned goals, including PDC specific language, afterward. No student mentioned parallel or distributed computing, problem decomposition, or event handling in their pre-course aspirations. After the course, 6 of 18 students (33%) spontaneously named these PDC concepts as things they had learned or wished to keep improving. For example, aspiring to "continue to improve my parallel and distributed computing skills," to "become better at problem decomposition," or noting "I improved on my understanding of parallel processing." This emergence of targeted vocabulary is direct qualitative evidence that the intervention shaped how students frame their own learning, and it aligns with the significant quantitative gains on the parallel-processing and event-handling items.

13.5.3 Did the PDC Intervention Improve Learning?

To study the impact of the PDC interventions discussed above on the learning of PDC concepts in a CS2 class, we compared student self-reported surveys from the baseline semester (Spring 2025) and the semester in which a PDC intervention was delivered (Spring 2026). Two questions are asked: (1) did students in the intervention year improve more on PDC measures than the baseline year, and (2) does the intervention appear in what students say, in their own words, that they learned? Table 13.19 reports wave means on the 1–7 scale. "Gain" is post − pre within a year; "DiD" is the 2026 gain minus the 2025 gain. Positive DiD favors the intervention.

Table 13.19: PDC competency items DiD analysis: baseline → intervention semester.

	Spring 2025			**Spring 2026**			
Item	Pre	Post	Gain	Pre	Post	Gain	**DiD**
Parallel processing	3.25	4.00	+0.75	3.95	4.80	+0.85	+0.10
Distributed computing	2.50	3.25	+0.75	3.45	4.32	+0.87	+0.12
Event handling	4.25	5.83	+1.58	3.00	4.96	+1.96	+0.38
Understanding of PDC	4.35	4.83	+0.48	4.65	5.36	+0.71	+0.23

The open ended response question was coded for unprompted mentions of PDC concepts (parallel, distributed, task decomposition, event handling, multi threading) over substantive responses. At entry, essentially no student in either cohort spontaneously referenced PDC. The divergence appears only at the post-course wave of the intervention year, where PDC mentions rose from 0/12 (2025 post) to 7/24 (2026 post). The results summarized in Table 13.20 show a statistically significant increase despite the small sample sizes.

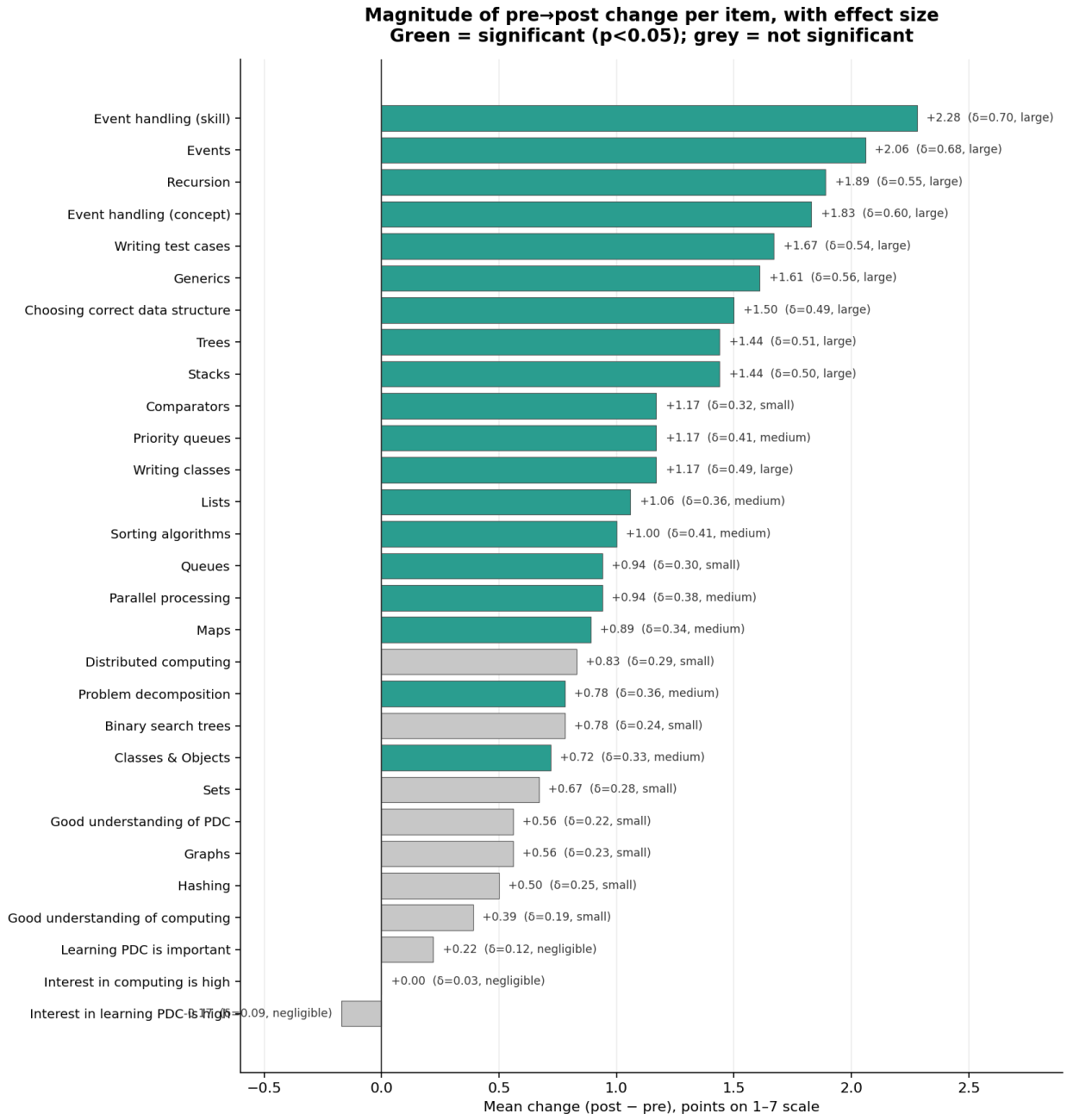


Figure 13.8: Pre to post shift for every survey item (n = 18). Mean change per item with Cliff's delta effect size. Green bars are significant ($p < 0.05$); grey are not.

Table 13.20: Unprompted PDC mentions in the open ended question (Q2).

Survey	PDC mentions	Substantive responses	%
2025 pre	1	20	5%
2025 post	0	12	0%
2026 pre	0	19	0%
2026 post	**7**	**24**	**29%**

13.6 Other Activities and Ideas

No additional activities were implemented during the PDC intervention semester in the CS2 course at USI.

13.7 Conclusion

Across the Spring 2026 offering of our CS2 course, we integrated a sequence of unplugged and plugged-in activities designed to introduce parallel and distributed computing (PDC) concepts alongside the traditional CS2 topics, such as data structures, object-oriented design, and event handling. To gauge their effect, we administered matched pre- and post-course experience surveys (19 pre-course and 24 post-course responses, yielding 16 matched pre/post pairs). The results, together with our own classroom observations, suggest that early, low-stakes exposure to PDC through tangible activities is both feasible in CS2 and modestly effective at building student confidence, while also surfacing clear boundaries on what a single semester can achieve.

- Lessons Learned
  - Embedding parallel and distributed computing into CS2 shifted self-reported experience in the intended direction on every PDC-related construct we measured. Among matched respondents, self-rated experience rose for parallel processing (+0.60 on our five-point experience scale), event handling (+0.88), problem decomposition (+0.53), and distributed computing (+0.36). The largest gains clustered around the concepts our activities targeted most directly, indicating that PDC ideas can be introduced at the CS2 level without displacing core content. For example, the "overly complicated machine" unplugged activity and the Greenfoot plugged in activity directly targeted the traditional CS2 concept, event handling.
  - The unplugged activities appear to have given students durable vocabulary for talking about the material. Unlike the pre-course survey, in the post-course survey, multiple students spontaneously cited "parallel computing," "parallel processing," and "event handling" as things they had learned and wished to continue developing.
  - Use of tangible unplugged activities before the programming driven plugged-in activities teaching the same concept appeared to lower the barrier to abstract concepts.

- Teaching Tips for Instructors
  - Introduce the unplugged activity before the corresponding code. Students who had physically modeled a parallel or event-driven process arrived at the plugged-in implementation with a mental model to map onto, rather than encountering the abstraction cold.
  - Anchor PDC in the CS2 concepts students already find familiar. That enables PDC to feel like an extension of CS2 rather than a foreign unit. This lowers resistance and reinforces prior material at the same time. This also allows instructors to introduce PDC concepts without displacing their original content.
- Inferences and Limitations
  - Our measures are self-reported and were not paired with a direct assessment of PDC knowledge. Gains in self-rated experience may reflect growth in confidence, calibration, or familiarity with terminology as much as growth in demonstrated skill, and self-report is subject to self-selection effects.
  - The study is small and uncontrolled. With 19 pre-course and 24 post-course responses and only 16 matched pairs from a single course, a single instructor, and a single institution, the findings should be read as descriptive.

13.7.1 Future Work

In the future iterations of the CS2 course at USI, we intend to grow the evidence base by replicating the intervention across multiple semesters and, where possible, additional sections, which would both increase statistical power and improve the significance testing. Finally, we plan to track longitudinal outcomes by following students into later courses to examine whether early CS2 exposure to PDC correlates with greater willingness to enroll in advanced PDC coursework. Together these efforts would move the work from a promising single-semester case study toward a more robust account of how unplugged and plugged-in activities shape students' trajectories in applying PDC.

Appendix

Github Link to Instructional Material

The USI CS2 repository containing relevant PDC intervention material as described in this chapter can be found at: *USI-CS2-PDC-Github*.

Detailed Pre and Post Syllabus

The Parallel and Distributed Computing (PDC) intervention was implemented in CS 358, Intermediate Object-Oriented Programming, the CS2 course at the University of Southern Indiana. CS 358 is a three-credit, Java-based course that builds on introductory OOP to cover intermediate object-oriented programming, including inheritance, polymorphism, interfaces, event-based programming, and data structures, using the textbook, Starting Out with Java: Control Structures through Objects (Pearson Revel Ed.) by Tony Gaddis. The pre-intervention syllabus followed a conventional CS2 structure, listing eleven course learning objectives centered on intermediate Java concepts, object-oriented principles, advanced Java features, design patterns, debugging, file I/O operations, multithreaded applications, group collaboration, code review and documentation, a comprehensive final project, and problem-solving, with no explicit treatment of parallel and distributed computing. The post-intervention syllabus preserved this structure but wove PDC directly into two existing objectives rather than adding a standalone one: under Advanced Java Features it added the implementation of event-handling concepts, and under Create Multithreaded Applications, it added exploring the challenges and best practices of concurrent programming and evaluating and comparing the execution performance of parallel programs. Under these objectives, students explain sequential versus parallel computing, data parallelism, speedup, and parallel overhead, and apply existing parallel methods to optimize sequential code. The accompanying course event calendar integrates the PDC interventions directly into the existing eight-module schedule, pairing an "unplugged" and a "plugged-in" activity for each concept. For event handling, an unplugged Overly Complicated Machine exercise and a plugged-in Greenfoot game is delivered as the group midterm project. For parallelism, an unplugged Fork/Join-on-Post-its demonstration of the parallel sum of an array of integers together with a plugged-in Java Fork/Join primality test along with a plugged-in Java ExecutorService parallel image-processing application is implemented as a final project. In this way, PDC concepts were woven into an otherwise standard CS2 sequence without displacing existing core course content.

Organization of Material

The `chapter13-USI` directory is organized as follows:

```
chapter13-USI/
  Detailed_Pre_and_Post_Syllabus/
  Evaluation_Data_and_Analysis/
  Plugged-In_Activities/
```

```
Unplugged_Activities/
USI-CS2_CourseCalendar_Spring2026.pdf
readme.md
```

Detailed_Pre_and_Post_Syllabus

Purpose: This directory (available at *Detailed-Pre-Post-Syllabus*) contains detailed syllabus materials for pre- and post-course assessments.

Contents: Documentation and resources related to course prerequisites and learning outcomes established before and after course instruction.

Usage: Reference materials for understanding the syllabus structure and course requirements.

Evaluation_Data_and_Analysis

Purpose: This directory (available at *Evaluation-Data-and-Analysis*) houses data collection and analytical materials used for course evaluation.

Contents: Assessment data, evaluation metrics, and analysis documentation for measuring student learning outcomes and course effectiveness.

Usage: Source for course evaluation reports, student performance metrics, and instructional efficacy analysis.

Plugged-In_Activities

Purpose: This directory (available at *Plugged-In-Activities*) contains computer-based instructional activities that require digital resources and technology.

Contents: Programming exercises, computational thinking labs, hands-on coding activities, and digital learning modules designed to develop technical competencies.

Usage: Classroom instruction materials for technology-integrated lessons and practical programming exercises.

Unplugged_Activities

Purpose: This directory (available at Unplugged-Activities) contains non-digital instructional activities focused on foundational concepts.

Contents: Paper-based exercises, kinesthetic learning activities, collaborative group work, and conceptual activities that develop problem-solving skills without direct technology use.

Usage: Complementary classroom materials for reinforcing core concepts and engaging diverse learning styles.

USI-CS2_CourseCalendar_Spring2026.pdf

Purpose: Official course calendar (available at *USI-CS2-Course-Calendar-Spring2026*) for the Spring 2026 CS2 course at the University of Southern Indiana.

Contents: Academic schedule including lecture dates, assignment deadlines, examination dates, and important institutional dates.

File Type: PDF document

Usage: Primary scheduling reference for students and instructors.

readme.md

Purpose: Quick reference documentation for the USI module.

Contents: Overview, navigation guide, and key information about accessing and utilizing resources in this directory.

Usage: Entry point for users to understand the module's organization and purpose.

Survey and Other Anonymized Data and Analysis

On the GitHub repository, the directory *Evaluation-Data-and-Analysis*, contains the de-identified and anonymized data and the analytical materials used for the evaluation of the PDC interventions.

Chapter 14

CS2 Course Package - University of Nebraska–Lincoln (UNL):

Minimal-Change Java Integration through Connection Pooling and Parallel Sorting[†]

Chris Bourke[1]

[1]University of Nebraska–Lincoln

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

14.1 Introduction . . . 408
14.2 How CS2 was changed and PDC topics integrated . . . 408
14.2.1 Integrated Syllabi (Pre and Post) . . . 408
14.2.2 Highlights . . . 408
14.2.3 Challenges . . . 408
14.3 New Unplugged Activities . . . 408
14.4 New Plugged-in Activities . . . 409
14.4.1 Connection Pooling . . . 409
14.4.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 409
14.4.1.2 Algorithms . . . 409
14.4.1.3 Implementation Details . . . 409
14.4.2 Parallel Sort Lab . . . 409
14.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 409
14.4.2.2 Algorithms . . . 410
14.4.2.3 Implementation Details . . . 410
14.4.2.4 Experience and Teaching Tips . . . 410
14.4.2.5 Data and Analysis . . . 410
14.5 Course-wide Pre and Post Course Surveys . . . 411
14.6 Other Activities and Ideas . . . 411
14.7 Conclusion . . . 412
Appendix . . . 413

Chapter tables

14.1 UNL CS2 Learning Outcomes & Bloom's Levels . . . 406
14.2 Survey Results for CS2 - UNL . . . 411

Abstract

A new lab was created and an old lab was updated in a CS2 course (CSCE 156) at the University of Nebraska–Lincoln in order to introduce PDC concepts. The revised lab introduces a parallel merge sort and has students benchmark it against several sequential sorting algorithms. The new lab introduces connection pooling in a multithreaded application to demonstrate speedup.

Descriptor Elements

Programming Language Employed: The labs utilize Java, JDBC, and SQL.

Relevant core courses: Computer Science 2 (CS2)

Pre-requisites: Students must have completed a CS1 or received credit for CS1 which includes mastery of at least one high-level programming language.

Relevant PDC topics:

Relevant PDC topics and Blooms levels Table 14.1 shows the activity-level learning outcomes added to UNL CS2 course with the incorporation of Parallel & Distributed Computing topics. Blooms levels are listed for each module and activity (Remembering (RE), Understanding (UN), Applying (AP), Analyzing (AN), Evaluating (EV), and Creating (CR)).

Table 14.1: UNL CS2 Learning Outcomes & Bloom's Levels

PDC Concept	**Module Content & Bloom's Level**	
	Connection Pooling 14.4.1	**Parallel Sorting 14.4.2**
Data Parallelism	Understanding (UN)	Understanding (UN)
Speedup and Benchmarking	Understanding (UN)	Understanding (UN)
Distributed Computing	Understanding (UN)	
Producer-consumer Pattern	Understanding (UN)	
Thread-safe Queue	Applying (AP)	

Learning outcomes:

1. Students should understand the complexities involved in a distributed system and how to handle those complexities.
2. Students should have an appreciation of the speedup that parallel computing can provide.

Context for use: The CS2 course at UNL (CSCE 156) covers Java (as a second programming language), OOP, SQL and databases, list-based data structures, stacks, queues, binary search trees, and heaps. There are several CS1 courses that feed into this course and so students have a variety of programming language background. This is a project-based course; most of the assessment is based on a prescribed database-backed project that is developed iteratively over the course of the semester.

14.1 Introduction

One new lab was created and one current lab was modified to introduce PDC concepts into a CS2 course at UNL. The new lab consisted of a connection pooling project and an old sorting lab was updated to include a parallel mergesort algorithm for benchmarking. Both of these are particularly relevant in the context of a CS2 course as databases and sorting are common topics for a CS2 course.

14.2 How CS2 was changed and PDC topics integrated

Minimal modifications were necessary. The sorting lab had already existed and code was provided to students to benchmark. The new lab was introduced in a week that had not had an associated lab for some time. A lab on recursion has been removed prior to the integration of PDC material as it duplicated CS1 material. These modifications presented minimal challenges as they simply enhanced existing topic and material.

14.2.1 Integrated Syllabi (Pre and Post)

No changes were made to the syllabus, PDC related activities were integrated as complementary to existing topics and activities.

14.2.2 Highlights

The course remained relatively unchanged. The two labs complemented and enhanced existing material and topics. One lab already existed and was extended which did not require removing existing content. The second lab was introduced into a week that there was no lab prior to the change.

14.2.3 Challenges

The major logistical challenge for adopters may be getting access to an SQL server instance for students. One workaround may be to adapt the lab to utilize a local SQLite instance that can be added to the Java project and eliminates the need for a remote server. However, some of the experience would be lost as the network lag (that contributes to the slowdown and subsequent speedup) is eliminated and SQLite is not as performant for distributed (write) connections.

14.3 New Unplugged Activities

No unplugged activities were developed or adopted. This course is run in a hybrid/asynchronous manner and has large enrollments, so unplugged activities are not easily integrated.

14.4 New Plugged-in Activities

14.4.1 Connection Pooling

14.4.1.1 Problem Description, Prerequisites, and Learning Outcomes

One of the primary learning outcomes of CS2 is familiarity with database design, SQL, and database connectivity. Typically students build a substantial database-backed applications, but these are usually single threaded applications.

To expose students to a more realistic distributed database system (as well as load testing and multithreaded applications), we developed a connection pooling lab. The lab uses multiple threads to insert random records into a remote database. To facilitate multiple threads, a connection pool is created using a blocking queue to store and reuse database connections. Students complete the lab by implementing code to retrieve and return database connections using the pool rather than opening/closing a new one every time. Students run several simulations with a configurable number of threads and random records in order to witness the speedup that multiple threads (and using connection pooling) offers.

The integration of this lab material may require previous exposure or familiarity with SQL, JDBC, and queues.

14.4.1.2 Algorithms

A consumer-producer pattern is introduced to manage multiple database connections in a thread-safe shared resource pool (Java's `BlockingQueue`). Thread workers are used to simulate a configurable load of inserting multiple records into a database.

14.4.1.3 Implementation Details

The lab writeup along with all source code is available on GitHub at https://github.com/cbourke/PDCPooling. The project contains necessary SQL files and instructions to setup the database. The repository itself is an Eclipse-based Java project. A few JAR file dependencies have been included in the project.

14.4.2 Parallel Sort Lab

14.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

Students are introduced to basic sorting algorithms in CS1 at UNL, but revisit these topics in CS2 in the context of algorithms and algorithm analysis.

Students complete several implementations of basic sorting algorithms (insertion sort, quick sort) as well as a custom Java `Comparator` to order geographic location objects. They then run several benchmarks and observe how much longer each algorithm takes to sort larger and larger sets of data. This lab was updated to include a parallel merge sort using Java's built-in `Arrays.parallelSort` which uses an internal fork-join mechanism to sort in parallel.

Prerequisites for this lab include mastery of lists and familiarity with comparators and sorting algorithms. A basic understanding of parallelization is also necessary. Using the built-in parallel sort can obscure implementation details but makes the lab accessible and more easily adoptable since no low-level fork-join knowledge is required.

14.4.2.2 Algorithms

Java's `Arrays.parallelSort` implementation is a hybrid merge sort algorithm. If the array is less than a certain threshold it defaults to Tim sort, otherwise it uses a `ForkJoin` common pool to break the array into subarrays and sort them in parallel.

14.4.2.3 Implementation Details

The lab writeups along with all source code is available on GitHub:

- https://github.com/cbourke/PDCPooling
- https://github.com/cbourke/CSCE156-Lab13

14.4.2.4 Experience and Teaching Tips

Both labs are heavy on concepts, but light on coding required of students, making these labs easily accessible to most CS2 students. For the connection pooling lab, however, students will need a prior background in SQL and access to an SQL database, ideally remotely in order to provide an observable difference when benchmarking.

14.4.2.5 Data and Analysis

The common pre/post surveys (See Section 10.5) were administered during the spring 2025 semester. The results are reported in Table 14.2. The pre/post survey results were paired (per-student) for analysis. With respect to relevant PDC topics, gains were significant and substantial for parallel and distributed computing (both topics covered by labs) while gains for event handling were negligible, reflecting that this topic was not covered in the course. With respect to other topics, gains were significant and substantial with all topics covered. There were small but statistically significant declines in Interest in Computing and Interest in PDC. We do not claim generalization and acknowledge the limitations of this data as it represents self-reported data and the sample size ($n = 33$) is relatively small.

Table 14.2: Survey Results for CS2 - UNL. Sample size was $n = 33$ and included data from spring 2025. The pre- and post means are reported along with the difference (Diff). Statistical significance is measured using a non-parametric Wilcoxon signed-rank (paired) test. The Wilcoxon statistic (W) is reported along with the p-value. Significance is indicated as Yes using $p < 0.05$. Effect size is reported using Cliff's Delta (δ). A negative value indicates an increase in the post-survey data. The interpretation of the effect size is reported (negligible, small, medium, large) as suggested by [55].

Item	Pre	Post	Diff	W	p-value	Sig?	Cliff's δ	Interp.
Computing Topics								
Classes/Objects	3.88	5.42	1.55	5.0	0.000	True	-0.52	large
Lists	3.79	4.91	1.12	21.0	0.000	True	-0.45	medium
Generics	1.76	4.30	2.55	3.5	0.000	True	-0.87	large
Stacks	2.36	4.39	2.03	26.5	0.000	True	-0.70	large
Queues	1.88	4.30	2.42	5.5	0.000	True	-0.80	large
Sorting Algos	3.42	4.67	1.24	3.5	0.000	True	-0.46	medium
Comparators	3.30	4.76	1.45	9.0	0.000	True	-0.45	medium
Recursion	3.70	4.91	1.21	25.0	0.000	True	-0.46	medium
Maps	2.09	4.97	2.88	3.5	0.000	True	-0.82	large
Sets	2.36	4.52	2.15	18.0	0.000	True	-0.69	large
Trees	2.52	4.33	1.82	24.0	0.000	True	-0.64	large
BST	2.39	4.12	1.73	30.5	0.000	True	-0.64	large
Priority Queues	1.64	3.79	2.15	13.5	0.000	True	-0.78	large
Hashing	1.61	3.52	1.91	25.0	0.000	True	-0.77	large
Graphs	2.76	4.00	1.24	105.5	0.003	True	-0.44	medium
Unit Tests	3.00	5.03	2.03	6.0	0.000	True	-0.62	large
Big-O	1.52	4.39	2.88	6.5	0.000	True	-0.89	large
DS Choices	3.03	4.55	1.52	18.0	0.000	True	-0.53	large
Designing/Writing Classes	3.27	5.52	2.24	5.5	0.000	True	-0.67	large
Decomposition	4.33	5.24	0.91	45.5	0.002	True	-0.33	medium
PDC Topics								
Parallel	2.45	3.39	0.94	75.5	0.005	True	-0.36	medium
Distributed	2.55	3.88	1.33	65.0	0.003	True	-0.45	medium
Event	3.24	3.48	0.24	163.5	0.754	False	-0.11	negligible
Attitudes								
Interest in Computing	6.27	5.79	-0.48	0.0	0.007	True	0.23	small
Interest in PDC	5.58	5.12	-0.45	30.0	0.041	True	0.16	small
Learning PDC is Important	6.03	5.76	-0.27	28.5	0.116	False	0.19	small
Understanding of Computing	4.88	5.21	0.33	81.5	0.065	False	-0.16	small
Understanding of PDC	3.42	4.55	1.12	56.5	0.002	True	-0.35	medium

14.5 Course-wide Pre and Post Course Surveys

See Section 10.5.

14.6 Other Activities and Ideas

Students could learn the same concepts using any consumer-producer pattern other than using it for database connection pooling. For example, any resource-intensive task (processing whole images, search, etc.) on a sufficiently large set of data lends itself to easy parallelization. Using a queue to store tasks and worker threads conveys the same ideas.

14.7 Conclusion

Though CS2 is less standardized than CS1, it is still possible to introduce PDC related concepts in a way that complements topic coverage using a minimal-change integration strategy. That is, a current course does not have to face significant disruption or modification in order to introduce PDC topics in a feasible manner. Most CS2 courses cover topics that easily lend themselves to adding PDC-adjacent concepts and topics. For example, CS2 courses typically cover queue data structures. It is an easy extension to demonstrate a use case for (thread safe) queues as an added PDC topic.

Appendix

Github Link to Instructional Material

The two labs are available as Eclipse-based Java projects on GitHub. They are author-maintained repositories. Each repository's `readme.md` file contains the lab writeup and instructions for students.

- Connection Pooling Lab: `https://github.com/cbourke/PDCPooling`
- Sorting Lab: `https://github.com/cbourke/CSCE156-Lab13`

Additional material can be found at the CDER GitHub repository for this chapter (here).

Detailed Pre and Post Syllabus

No changes were made to the syllabus.

Organization of Material

Each GitHub repository has a `readme.md` file at the top level which serves as both a primer and handout with instructions to students for setup and activities. Both are Eclipse-based Java projects.

Survey and Other Anonymized Data and Analysis

See Section 10.5 for Survey content and Section 14.4.2.5 for results and analysis.

Chapter 15

CS2 Course Package - Webster University:

C++ Infusion through Parallel Merge Sort, OpenMP, and Performance Analysis[†]

Peter Maher[1]

[1]Webster University

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

15.1 Introduction . . . 418
15.2 How CS2 was changed and PDC topics integrated . . . 419
15.2.1 Integrated Syllabi . . . 419
15.2.2 Highlights . . . 419
15.2.2.1 Unplugged Activities . . . 420
15.2.2.2 Programming Demo . . . 420
15.2.2.3 Introduction to Parallel Programming . . . 421
15.2.2.4 Threads . . . 421
15.2.2.5 Shared Memory . . . 421
15.2.2.6 Message Passing . . . 421
15.2.2.7 Basic Principles of Parallel Computing . . . 422
15.2.2.8 Merge Sort Algorithm . . . 422
15.2.2.9 Implementation Details . . . 422
15.2.2.10 Experience and Teaching Tips . . . 424
15.2.2.11 Slides in the CS2 Course . . . 424
15.2.2.12 Data and Analysis . . . 427
15.2.3 Changes in the Course . . . 427
15.2.3.1 Assignments . . . 427
15.2.3.2 Integration of PDC Topics into the CS2 Course . . . 429
15.3 New Plugged-in Activities . . . 429
15.4 Course-wide Pre and Post Course Surveys . . . 431
15.5 Other Activities and Ideas . . . 432
15.6 Conclusion . . . 433
Appendix . . . 435

Chapter tables

15.1 Integration of PDC topics into the CS2 course. . . . 430
15.2 Descriptive COSC 3100 pre- and post-survey summaries. The samples are not matched. 431
15.3 Comparison between serial and OpenMP parallel merge sort implementations. . . . 434
15.4 Descriptive summary of selected COSC 3100 survey responses. Pre- and post-survey samples are not matched. . . . 437

Abstract

This chapter describes a lightweight integration of Parallel and Distributed Computing (PDC) concepts into a CS2-level C++ course at Webster University, a primarily undergraduate institution. The existing course emphasizes recursion, sorting, object-oriented programming, trees, graphs, and algorithmic problem solving. Rather than replacing these core topics, PDC concepts were introduced through existing CS2 material, particularly sorting and performance measurement. Students first review the motivation for parallel computing and the relationship between task decomposition and concurrency, then examine serial and parallel implementations of merge sort using OpenMP and measure execution time using the C++ `chrono` library. The instructional goal is not to provide a comprehensive course in parallel programming, but to help students recognize opportunities for parallelism, understand that parallel execution introduces overhead and coordination requirements, and interpret measured performance rather than assuming that parallel code is always faster.

Programming Language Employed: C++

Relevant core courses: CS2, Data Structures, Object-Oriented Programming

Pre-requisites: CS1. Students entering the course are expected to have prior experience with C++ programming fundamentals, including functions, arrays, basic classes, control structures, and algorithmic problem solving. The PDC material builds on these skills rather than requiring prior coursework in operating systems, computer architecture, or parallel programming.

Textbook: *Starting Out with C++ from Control Structures to Objects*, 10th edition, Tony Gaddis [23].

Relevant PDC topics:

PDC Concept	Bloom's Level
Motivation for Parallel Computing	Understanding (UN)
Task Decomposition	Applying (AP)
Shared-Memory Parallelism with OpenMP	Applying (AP)
Threads and Concurrent Execution	Understanding (UN)
Synchronization and Coordination	Understanding (UN)
Performance Measurement and Benchmarking	Applying (AP)
Speedup and Parallel Overhead	Understanding (UN)
Serial versus Parallel Algorithm Comparison	Applying (AP)
Distributed-Memory / Message-Passing Concepts	Remembering (RE)

Learning outcomes:

1. Discuss and apply good software development practices involving design, coding, and testing.
2. Demonstrate understanding of recursive functions.

3. Describe and compare major searching and sorting algorithms.
4. Compare and critically assess algorithms for specific applications.
5. Describe and use binary trees.
6. Demonstrate an understanding of graphs and their applications.
7. Explain the purpose of parallel programming and identify portions of an algorithm that may be executed concurrently.
8. Measure and compare the execution time of serial and parallel implementations.
9. Interpret observed performance differences in terms of useful computation, coordination, and parallel overhead.

Context for use: Webster University is a primarily undergraduate institution located in St. Louis. The Department of Computer and Information Sciences offers several undergraduate computing programs. Within the CS2 sequence, COSC 3100 develops students' C++ programming and data-structure knowledge through topics such as recursion, sorting, classes, binary trees, heaps, AVL trees, Huffman coding, graphs, and shortest-path algorithms. Typical enrollment is relatively small, which makes guided demonstrations and discussion practical. The PDC integration described in this chapter was designed to fit within the existing course rather than operate as a separate parallel-computing unit.

15.1 Introduction

Parallel and distributed computing is often introduced after students have completed courses in data structures, algorithms, computer architecture, or operating systems. However, many of the underlying ideas can be introduced earlier when they are connected to algorithms that students already understand. In a CS2 course, sorting provides a useful entry point because students can first reason about a familiar serial algorithm and then ask whether independent pieces of that computation can be executed concurrently.

The goal of this course modification was therefore modest: to expose students to parallel thinking without displacing the core CS2 curriculum. The instructional sequence connects three ideas that students can observe directly: decomposition, execution time, and performance comparison. Students examine how a problem can be divided into independent subproblems, how C++ code can express parallel execution through OpenMP, and how measured runtime can be used to evaluate whether the parallel version actually improves performance.

A basic timing pattern using the C++ `chrono` library is shown in Listing 15.1. This code is deliberately simple because the purpose is to give students a reusable measurement pattern that can be applied to multiple algorithms.

Listing 15.1: Basic C++ code for measuring the execution time of a sorting algorithm.

```
#include <chrono>
#include <iostream>

using namespace std;
using namespace chrono;

auto start = high_resolution_clock::now();

// Sorting process

auto stop = high_resolution_clock::now();

auto sortDuration =
    duration_cast<microseconds>(stop - start);

cout << "Sort timing: "
     << sortDuration.count()
     << " microseconds" << endl;
```

The timing exercise is important pedagogically because it changes the question from "Does the program work?" to "How does the program behave, and why?" Students are encouraged to compare implementations empirically rather than assuming that parallel execution must always be faster. This is particularly important because parallel code introduces overhead associated with creating and scheduling work, coordinating threads, and waiting for dependent computations to complete.

15.2 How CS2 was changed and PDC topics integrated

The overall structure of COSC 3100 remained largely unchanged. PDC was integrated into existing algorithmic topics rather than introduced as a separate, advanced systems unit. Sorting was the primary context because students already study recursive and comparison-based sorting algorithms in CS2. Performance measurement was then added so that students could compare a serial implementation with a parallel implementation using measured execution time.

The transformation followed a progression from concept to implementation:

1. review a serial algorithm that students already understand;
2. identify independent portions of the computation;
3. discuss why those portions might execute concurrently;
4. introduce the OpenMP mechanism used to express parallel work;
5. measure serial and parallel execution time; and
6. interpret the result, including cases in which parallel overhead reduces or eliminates a speed advantage.

This approach intentionally limits the amount of new syntax introduced at one time. Students are not expected to master the complete OpenMP programming model. Instead, OpenMP is used as a concrete mechanism for expressing concepts such as task decomposition, concurrent execution, and synchronization.

15.2.1 Integrated Syllabi

The general CS2 syllabus was preserved. PDC material was inserted into weeks in which sorting, algorithm timing, large datasets, and graph algorithms were already being discussed. The available course documentation indicates that PDC-related material appeared in several parts of the semester rather than replacing a single block of existing content.

The integration was designed to preserve core CS2 learning outcomes. Students still study recursion, sorting, object-oriented programming, trees, and graphs. The principal change is that selected algorithms are now accompanied by questions about decomposition and performance. For example, when merge sort is introduced, students can identify that the left and right halves are independent until the merge step. When timing is introduced, students can compare the serial and parallel versions rather than treating runtime as an abstract concept.

Because the implementation evolved across offerings, the schedule in Table 15.1 should be interpreted as the documented instructional pattern rather than as evidence that every activity was delivered identically in every semester.

15.2.2 Highlights

The strongest feature of the integration is the use of existing CS2 algorithms as the vehicle for PDC. Students do not need to learn an unfamiliar scientific application before they can reason about parallelism. Instead, they start with an algorithm that is already part of the course and consider where independent work exists.

A second emphasis is performance measurement. The course does not present parallel programming simply as an additional C++ syntax feature. Students are shown that performance must be measured and interpreted. A parallel implementation can outperform a serial implementation, but the result depends on workload size, available hardware, overhead, and the structure of the algorithm.

A third feature is the limited scope of the intervention. The course does not attempt to cover advanced PDC topics such as detailed scalability modeling, NUMA optimization, or distributed-memory programming in depth. Message passing and distributed computing may be introduced conceptually, but the implemented programming example focuses on shared-memory parallelism with OpenMP. This distinction is important for accurately describing what students actually experienced.

Modern generative AI tools also affect the design of programming assignments. Short code-generation tasks alone provide limited evidence that students understand the computational ideas. For that reason, the more useful PDC deliverables require students to run experiments, record timing results, compare implementations, and explain observed behavior.

15.2.2.1 Unplugged Activities

No dedicated, formally assessed unplugged PDC activity was retained in the available materials for this implementation. Conceptual examples and instructor-led discussion were used to motivate decomposition and concurrent execution, but these should not be represented as a separately evaluated instructional intervention.

For future offerings, an unplugged activity could be added before OpenMP instruction to make task decomposition more concrete. Such an activity should be clearly identified as a future or recommended addition unless it is actually implemented and documented. Maintaining this distinction is important when reporting course transformations.

15.2.2.2 Programming Demo

After the conceptual introduction, OpenMP is presented as a practical mechanism for expressing parallel work in C++. One challenge for students is the unfamiliar syntax of compiler directives, particularly `#pragma`. Rather than presenting `#pragma omp` as a special command to memorize, it is described as an instruction to the compiler that identifies how a region of code should be executed.

The demonstration begins with serial code and asks students to identify independent work before any OpenMP directive is introduced. This ordering emphasizes that parallelism is an algorithmic property first and a programming-language expression second.

For recursive merge sort, OpenMP tasks provide a natural mapping between the algorithm and the parallel implementation. The two recursive calls can be represented as separate tasks, while a synchronization point is required before the merge step. The demonstration therefore connects three ideas in one example: decomposition, task creation, and synchronization.

15.2.2.3 Introduction to Parallel Programming

Parallel programming allows multiple computational resources to contribute to the solution of a problem at the same time. A useful parallel solution generally begins by identifying work that can be separated into independent or partially independent tasks.

Students are introduced to the idea that parallel execution is not automatically beneficial. If a problem is too small, the overhead of creating and coordinating parallel work may exceed the time saved by concurrent execution. Similarly, if one portion of a computation depends on another, synchronization may limit the amount of useful concurrency.

In this course, the learning objective is therefore not merely to define parallel programming, but to connect the definition to an algorithm students can inspect. Merge sort is useful because the two recursive halves can be sorted independently, while the final merge cannot occur until both halves have completed.

15.2.2.4 Threads

Threads are units of execution within a process. Multiple threads can execute different portions of a program concurrently and can share access to the same address space. This shared-memory model makes threads useful for parallelizing work on a multicore processor.

Students are introduced to threads conceptually in order to understand what OpenMP is coordinating. The course does not require a comprehensive treatment of manual thread management with `std::thread`. Instead, OpenMP is used to reduce syntactic complexity while preserving the conceptual distinction between serial execution and work performed by multiple threads.

15.2.2.5 Shared Memory

In a shared-memory programming model, multiple threads can access common data in the same process. OpenMP supports this model through compiler directives and runtime functions.

Shared memory simplifies communication because threads do not need to send explicit messages to access common arrays or vectors. However, shared access also creates correctness concerns. If multiple threads modify the same data at the same time without appropriate coordination, race conditions may occur. In the merge-sort example, students can reason about why sorting separate array regions is safe during the recursive phase and why the merge operation must wait until the corresponding recursive tasks are complete.

15.2.2.6 Message Passing

Message passing is introduced as a contrasting PDC model. In distributed-memory systems, processes do not normally share a single address space and therefore communicate by explicitly sending and receiving data. MPI (*Message Passing Interface*) is a widely used standard for this model.

The available implementation materials do not document an MPI programming assignment in COSC 3100. Accordingly, message passing is treated here as conceptual exposure rather than as a programming competency assessed in the course. The distinction between

shared-memory OpenMP programming and distributed-memory message passing helps students see that "parallel computing" includes more than one execution model.

15.2.2.7 Basic Principles of Parallel Computing

The introductory discussion emphasizes a small set of principles that can be reused across algorithms:

- A computational problem must contain work that can be separated into tasks or data regions.
- Some of that work must be executable concurrently for parallelism to be useful.
- Dependencies among tasks determine where synchronization is required.
- Parallel execution introduces overhead, so additional computational resources do not guarantee proportional speedup.
- Performance claims should be supported by repeated measurement rather than by a single timing result.

The last point is particularly important in an introductory course. A single measured runtime can be affected by system load, input size, compiler settings, cache behavior, and other factors. Students should therefore view timing as an experiment. Even a simple exercise becomes more informative when the program is executed repeatedly and results are compared across multiple input sizes.

A basic speedup metric can be introduced without requiring advanced mathematics:

$$S_p = \frac{T_1}{T_p},$$

where T_1 is the execution time of the serial program and T_p is the execution time using p threads. Students can then interpret whether the measured speedup is greater than 1 and discuss why it may be substantially smaller than the number of threads used.

15.2.2.8 Merge Sort Algorithm

Merge sort is a recursive divide-and-conquer algorithm. A vector is repeatedly divided into smaller portions until the base case is reached. The smaller portions are then merged back together in sorted order.

15.2.2.9 Implementation Details

The serial recursive structure is shown in Listing 15.2. The two recursive calls operate on different portions of the vector and therefore provide a natural point for discussing task-level parallelism.

Listing 15.2: Serial recursive merge sort implementation.

```
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
```

```
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}
```

The merge function combines two sorted portions of the vector:

Listing 15.3: Merge operation used by serial and parallel merge sort.

```
void merge(vector<int>& arr, int left,
           int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];

    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ?
                   L[i++] : R[j++];

    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];
}
```

For the parallel version, OpenMP tasks provide a cleaner correspondence to the recursive structure than recursively creating nested parallel regions. A depth threshold is used so that very small recursive calls do not create excessive task overhead.

Listing 15.4: OpenMP task-based parallel merge sort.

```
void parallelMergeSort(vector<int>& arr,
                       int left,
                       int right,
                       int depth = 0) {
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;

    if (depth < 4) {
        #pragma omp task shared(arr)
        parallelMergeSort(arr, left, mid,
                          depth + 1);
```

```
        #pragma omp task shared(arr)
        parallelMergeSort(arr, mid + 1, right,
                          depth + 1);

        #pragma omp taskwait
    }
    else {
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
    }

    merge(arr, left, mid, right);
}
```

The task-based function should be invoked from a single OpenMP parallel region:

Listing 15.5: Calling the OpenMP parallel merge sort.

```
#pragma omp parallel
{
    #pragma omp single
    {
        parallelMergeSort(
            data, 0, data.size() - 1);
    }
}
```

Students can time both versions using the same measurement method. One classroom example in the existing material reported approximately 0.0241 seconds for regular merge sort and 0.0178 seconds for the parallel version. For that individual run, the observed speedup would be approximately

$$\frac{0.0241}{0.0178} \approx 1.36.$$

This result should be treated as an illustration rather than as general performance evidence. A stronger student experiment repeats the measurement, varies input size and thread count, and reports a summary statistic such as the median runtime. Students can then explain why the measured speedup is not equal to the number of threads.

15.2.2.10 Experience and Teaching Tips

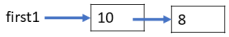
Providing clarification about the benefits of Parallel Programming is very important. The new assignments are shown in the next section of this chapter. Getting students to develop an algorithm that can be faster when using Parallel Programming does provide confirmation.

15.2.2.11 Slides in the CS2 Course

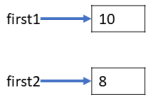
When covering the concepts of the Merge Sort algorithm, some slides are shown to the students to indicate some of the main topics. It is important to clarify how the algorithm is executed.

Merge Sort

- Assume that these two values are stored in a linked list:

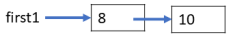


- The first step in the Merge Sort algorithm will split the two nodes into separate linked lists:

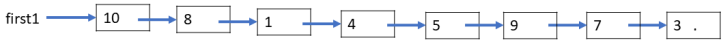


Merge Sort

- It will then determine which is the smaller value in the two nodes, and then add them together in order in a linked list:

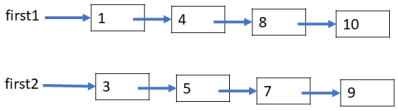


- Now assume that these values are stored in a linked list:



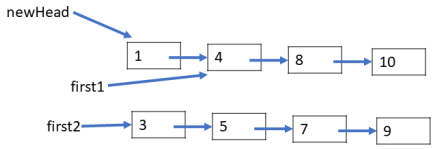
Merge Sort

- The Merge Sort algorithm will now divide the linked list into two linked lists, and will sort each value in order:

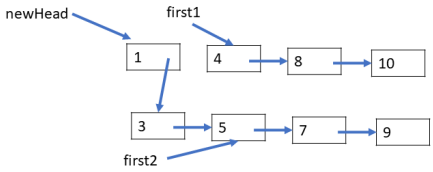


Merge Sort

- Then another pointer, ‘NewHead’, will point to the smaller of the current head node’s value. In this example, ‘NewHead’ will be pointing to the node with ‘1’, and ‘first1’ will be set to the next node in that linked list:



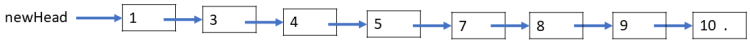
Merge Sort



- Compare the values in the notes pointed to by ‘first1’ and ‘first2’ again and make the same updates

Merge Sort

- This process will continue until ‘first1’ or ‘first2’ is set to the end of the linked list. Then the linked list will be in order:



15.2.2.12 Data and Analysis

The common pre/post surveys were administered during the Fall 2025 semester. The pre/post survey results were requested for analysis in the COSC 3100 course. The plan is that students will understand the concepts of Parallel Programming because it has been introduced in the C1 course that they have taken, and certainly the possible reason for using it in some algorithms. All students did confirm that they wanted to be learning more about Parallel Programming. As we move forward and certainly include Parallel Programming in this course, we are confident that it will become very familiar to all students. After graduating from the Computer Science degree, fully understanding the benefits of Parallel Programming will be very helpful as they begin to work in a Computer Science company.

15.2.3 Changes in the Course

The PDC integration was designed as an enhancement rather than a replacement of the CS2 curriculum. The principal changes were:

- **Performance was made explicit.** Students were asked to measure algorithm execution time rather than considering correctness alone.
- **Sorting became a context for decomposition.** Merge sort was used not only as a recursive sorting algorithm but also as an example of independent subproblems.
- **OpenMP provided a concrete shared-memory implementation.** Students could connect the conceptual idea of concurrency to a small amount of C++ code.
- **Interpretation was emphasized.** Students were encouraged to explain why a parallel version did or did not improve performance.
- **An assignment for Sorting code.** Sorting algorithms are always covered in this course, but now Parallel Programming is being used to improve the speed of Merge Sort.

15.2.3.1 Assignments

The following assignment is a requirement to write code to develop several Sorting algorithms. This will make students fully understand the way in which various sorting algorithms are implemented.

COSC 3100 – Data Structures II

Assignment 3

Develop a template class called 'SortedList' to store an array of values of some type, and provide the following public functions:

SortedList(int size)

Constructor in which the initial size of the array is specified and dynamically allocated. Pass the value 10 for the purpose of this assignment.

~SortedList()

Destructor to destroy the array of values.

void insert(const T& item);

Insert a value into the list. If the list is already full then an additional 10 elements should be added. *Make sure that this update of size is executed at least once.*

void randomise()

Randomise the order of the values in the array.

void quickSortA()

Perform the 'quick sort' algorithm to sort the array of values into ascending order.

void quickSortD()

Perform the 'quick sort' algorithm to sort the array of values into descending order.

void mergeSortA()

Perform the 'merge sort' algorithm to sort the array of values into ascending order.

void mergeSortD()

Perform the 'merge sort' algorithm to sort the array of values into descending order.

void display() const

Display the list of values.

Write a main() function to create an object of type SortedList. The object should be able to store values of type Stock. Populate the list with values by reading from a file and display the original list. Call the 'randomise()' function and re-display the list. Proceed by invoking one of the sorting functions, displaying the resulting list of values, and then calling 'randomise()'. Repeat these steps for the other sorting functions.

COSC 3100 – Data Structures II

Assignment 4

An assignment to determine the time taken for Sorting algorithms, and use Parallel programming. Determining the time taken for each Sorting algorithm is important to learn. Then determining the time taken for the Merge Sort algorithm when using Parallel Programming will provide clarification of the advantage. For each of the Sorting functions, determine the time taken to complete the process that they are providing.

SortedList(int size)

~SortedList()

void insert(const T& item);

void randomise()

void quickSortA()

void quickSortD()

void mergeSortA()

void mergeSortD()

void display() const

Then create additional functions for the MergeSort algorithm that use Parallel Programming:

void mergeSortAPP()

void mergeSortDPP()

Then determine the time taken to complete the process that they are providing. Compare the difference between the original algorithms and the algorithms using Parallel Programming. Use these times taken to clarify the advantages of Parallel Programming.

Some less essential repetition in existing examples can be reduced when instructional time is limited, but no core CS2 topic needs to be removed solely to introduce this material. The intended instructional footprint is small enough to be embedded within existing sorting and algorithm-analysis lessons.

15.2.3.2 Integration of PDC Topics into the CS2 Course

Table 15.1 summarizes the documented integration of PDC concepts into existing CS2 topics. The table distinguishes between conceptual exposure and activities that can produce a concrete student deliverable.

A reproducible version of the core activity can require the following student deliverable: serial and parallel runtimes for at least two input sizes, the number of threads used, calculated speedup, and a short written interpretation of whether parallel execution helped and why. This deliverable is recommended because it provides stronger evidence of understanding than code submission alone.

15.3 New Plugged-in Activities

The principal plugged-in activity uses C++ timing and OpenMP to compare serial and parallel merge sort. The activity can be completed as a guided laboratory or as a short programming assignment.

A suggested sequence is:

Table 15.1: Integration of PDC topics into the CS2 course.

Week	Activity	Time	CS2 Topics	PDC Topics
3	Introduce parallel-computing motivation and identify decomposition opportunities in familiar algorithms	1–2 lectures	Arrays, recursion, sorting	Parallelism, decomposition, dependencies, overhead
5	Measure execution time of several algorithms	1 lecture	Algorithm comparisons	Benchmarking and runtime measurement
7	Compare the time taken on several search algorithms	1–2 lectures	Recursive sorting algorithms	Work on determining which search algorithm might be the fastest
12	Use Parallel Programming for Sorting Algorithms	1–2 lectures	Sorting larger collections	Workload size, performance variability, overhead
15	Discuss whether graph problems contain parallelizable work	1 lecture or discussion	Graphs and shortest paths	Identifying potential concurrency; conceptual extension

1. Compile and run the serial merge-sort program.
2. Confirm correctness using a small dataset.
3. Measure runtime for a larger dataset using `chrono`.
4. Compile the OpenMP version with the required compiler flag.
5. Run the parallel implementation using a specified number of threads.
6. Repeat each timing several times.
7. Calculate speedup using $S_p = T_1/T_p$.
8. Explain whether the measured result supports the expectation that parallel execution is faster.

The student deliverable should include more than source code. A compact experimental table can be required:

Input Size	Threads	Serial Time	Parallel Time	Speedup

Students should also answer short interpretation questions such as:

1. Which portions of merge sort can execute independently?
2. Why is a synchronization point needed before the merge step?
3. Did the parallel implementation always outperform the serial implementation?
4. What sources of overhead could explain the observed results?

This activity addresses a limitation of code-only assessment. A student who can generate a parallel program but cannot explain the timing behavior has not necessarily demonstrated understanding of the underlying PDC concepts.

The original implementation included quiz questions related to parallel programming, but item-level quiz results suitable for systematic analysis were not available in the supplied course records. Therefore, this chapter does not use the quiz as quantitative evidence of learning. Future offerings should retain item-level results for a small set of PDC questions aligned directly with the learning outcomes.

15.4 Course-wide Pre and Post Course Surveys

Pre- and post-course survey files were examined to determine what conclusions can be supported by the available data. The pre-survey file contains nine responses, all of which identify COSC 3100 among the respondent's enrolled courses. The post-survey file contains 21 total responses from multiple courses; ten responses identify COSC 3100. Only those ten records are included in the COSC 3100 post-survey summaries reported here.

The survey records do not form a clean matched pre/post sample. Pre-survey responses were recorded in January–February 2025 and August 2025, while COSC 3100 post-survey responses were recorded in January 2025, December 2025, and April 2026. Names were optional, and only a small subset can be linked across files. These timing and matching limitations prevent a defensible student-level pre/post analysis. Accordingly, the values below are descriptive summaries of two available COSC 3100 response groups and should not be interpreted as learning gains caused by the PDC intervention.

For descriptive purposes, experience items were coded from 1 ("No Experience At All") to 7 ("Extremely Experienced"). Agreement items were coded from 1 ("Strongly Disagree") to 7 ("Strongly Agree"). Table 15.2 reports means for the nine pre-survey records and ten COSC 3100 post-survey records.

Table 15.2: Descriptive COSC 3100 pre- and post-survey summaries. The samples are not matched.

Survey Item	**Pre Mean (n=9)**	**Post Mean (n=10)**
Problem decomposition experience	4.00	4.10
Parallel-processing experience	2.78	3.00
Distributed-computing experience	2.44	2.60
Interest in computing	5.56	5.80
Interest in learning PDC	5.33	5.30
Belief that learning PDC is important	5.89	5.80
Perceived understanding of computing	4.89	5.20
Perceived understanding of PDC	4.22	4.20

The distributions provide additional context. In the pre-survey, five of nine respondents reported either no experience or very little experience with parallel processing. In the COSC 3100 post-survey subset, five of ten respondents reported either no experience or very little experience with parallel processing. For distributed computing, five of nine pre-survey respondents and seven of ten post-survey respondents reported either no experience or very little experience. These responses do not support a claim that the course produced broad mastery of PDC.

At the same time, students generally viewed PDC as worthwhile. On the pre-survey, eight of nine respondents selected partially agree, agree, or strongly agree for the statement that learning PDC is important. In the COSC 3100 post-survey subset, nine of ten respondents selected partially agree, agree, or strongly agree, with the remaining respondent selecting neutral. Interest in learning PDC was positive but somewhat more mixed in the post-survey group.

The post survey also asked students to rate several general COSC 3100 learning goals. Among COSC 3100 respondents with non-missing answers, mean self-ratings were 6.00/7 for

applying software-development practices, 6.11/7 for understanding and applying recursive functions, 5.56/7 for critically assessing algorithms, 5.78/7 for describing and using binary trees, and 5.00/7 for understanding and applying graphs. These items characterize students' perceptions of the overall course but do not directly measure PDC learning.

Several conclusions follow from this analysis. First, the original statement that students reached approximately 95% understanding of parallel programming is not supported by the supplied survey data and is therefore not reported. Second, because the samples are not matched and span multiple collection periods, differences between the means should not be interpreted as statistically or causally meaningful. Third, the surveys indicate that students value PDC but continue to report limited hands-on experience. This pattern supports the need for a more explicit programming activity and a dedicated assessment aligned with the PDC learning outcomes.

Future data collection should use a consistent course identifier, administer pre- and post-surveys within the same offering, create an anonymous matching code that permits student-level pairing, and retain results from dedicated PDC assessment items. A short concept assessment could ask students to identify parallelizable work, explain a synchronization requirement, calculate speedup from timing data, and interpret why a parallel program might fail to achieve linear speedup.

15.5 Other Activities and Ideas

The current implementation demonstrates that PDC can be introduced in CS2 without redesigning the entire course, but it also identifies several opportunities for improvement.

First, the merge-sort activity can be converted from an instructor demonstration into a structured laboratory in which every student produces timing data and a short interpretation. This would create a direct student artifact that can be assessed with a simple rubric.

Second, future offerings could compare more than one algorithm. Searching, graph traversal, or other divide-and-conquer problems could be used to ask whether parallelization is appropriate. Importantly, not every algorithm should be presented as a good candidate for parallel execution. Asking students to explain why an algorithm is difficult to parallelize can be as educational as implementing one that scales well.

Third, assessment should be aligned explicitly with the PDC learning outcomes. A future assessment could contain four components: identifying independent tasks, explaining race conditions or synchronization, calculating speedup, and interpreting measured performance. Retaining item-level pre/post results would make it possible to evaluate learning more directly than the general attitude survey used in the current implementation.

Finally, the distinction between CS1 and CS2 should remain intentional. CS1 can provide conceptual exposure to decomposition and parallel thinking, while CS2 can use students' greater experience with recursion, sorting, and data structures to support implementation and performance analysis. When students have already completed a PDC activity in CS1, the CS2 version should extend rather than duplicate that experience.

15.6 Conclusion

This course transformation illustrates a limited but practical way to introduce PDC in an existing CS2 curriculum. Rather than adding a large collection of advanced parallel-computing topics, the implementation uses material students already encounter—particularly recursion, merge sort, and algorithm timing—to introduce task decomposition, shared-memory parallelism, synchronization, and performance measurement.

The available evidence does not justify claims of large learning gains. The supplied survey files contain small, unmatched samples collected across multiple periods, and students' self-reported PDC experience remains limited. These limitations are important findings because they clarify what must be improved in future implementations: more structured hands-on work, explicit student deliverables, dedicated assessment items, and matched data collection.

The instructional value of the approach is therefore best understood as feasibility and curriculum integration rather than demonstrated effectiveness. CS2 students can engage meaningfully with parallel thinking when the material is connected to familiar algorithms, and a small OpenMP activity can shift attention from merely writing correct code toward measuring, comparing, and explaining program behavior.

Table 15.3: Comparison between serial and OpenMP parallel merge sort implementations.

Step	**Serial Merge Sort**	**Parallel Merge Sort**
Data Preparation		
	`vector<int> data1;` `readFile(data1);`	`vector<int> data2 = data1;`
Start Timer		

	`auto start =`
	`  chrono::high_resolution_clock::`
	`  now();`
		`auto start =`
		`  chrono::high_resolution_clock::`
		`  now();`

Sorting		

	`mergeSort(data1, 0,`
	`  data1.size()-1);`
		`#pragma omp parallel`
		`{`
		`  #pragma omp single`
		`  {`
		`  parallelMergeSort(`
		`  data2, 0,`
		`  data2.size()-1);`
		`  }`
		`}`

Synchronization	Not required between independent recursive calls because execution is serial.	
		`#pragma omp taskwait`
		before merging the two sorted halves.
End Timer		

	`auto end =`
	`  chrono::high_resolution_clock::`
	`  now();`
		`auto end =`
		`  chrono::high_resolution_clock::`
		`  now();`

Execution Time		

	`double serialTime =`
	`  chrono::duration<double>(`
	`  end-start).count();`
		`double parallelTime =`
		`  chrono::duration<double>(`
		`  end-start).count();`

Performance Interpretation	Baseline runtime T_1.	Calculate $S_p = T_1/T_p$ and explain the observed speedup and possible overhead.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter15-Webster

Detailed Pre and Post Surveys

Introduction to Programming	Introduced the concept of parallel computing through unplugged activities and real-world examples of concurrency.
Loops and Iteration	Discussed independent iterations, data parallelism, and dividing repetitive tasks among multiple workers.
Arrays, Vectors, and Strings	Demonstrated data decomposition and partitioning of large datasets into smaller independent sections.
Searching and Counting Problems	Compared sequential and parallel approaches using classroom activities and simple performance measurements.
Grid-Based Simulations	Used zombie spread and image-processing style activities to illustrate parallel updates on large datasets. (concepts, programming demo, programming assignments)
Performance Measurement	Introduced basic timing experiments, speedup, and efficiency using small C++ examples.

Organization of Material

The instructional package provides four documents, showing how the benefits of Parallel Programming is now being included in the C++ programming language.

1. **Exam Questions**
   Some general questions given in an Exam that will clarify that students do understand the benefits of Parallel Programming, and know how to write it is the C++ programming language. There is a question about how to write a function for a sorting algorithm, then another function that will use Parallel Programming to do the sorting process.
2. **Idea of Parallel Programming**
   Slides that introduce the meaning of Parallel Programming. There is an example of how 'shared memory' can be used to achieve a requirement. Also, how the C++ programming language can use Parallel Programming to complete something faster.
3. **Parallel Programming with Merge Sort**
   The Merge Sort algorithm is introduced, and showing how it can be implemented in the C++ programming language. Then using Parallel Programming in the Merge Sort

algorithm is included. This clearly provides an example of how Parallel Programming can be coded.

4. **Quiz Questions**
   Making sure that the students know the benefits of Parallel Programming, what is the meaning of "Treads", and then confirming that determining the time taken for a function to be executed can be determined. The timing of a function is important to determine when using Parallel Programming code.
5. **Quiz Questions**
   In the C++ programming language, using a 'Library' will be necessary for Parallel Programming. And how "Threads" are used in the C++ programming language is another part of using Parallel Programming.

Survey and Other Anonymized Data and Analysis

Pre- and post-course surveys were used to collect descriptive information about students' prior computing experience, exposure to parallel and distributed computing (PDC), interest in PDC, and self-reported understanding of computing concepts. The survey data are used in this chapter to characterize the students and their perceptions of PDC; they are not treated as a controlled or matched pre/post assessment of learning.

Data included in the analysis. The available pre-survey dataset contains nine responses, all of which identify COSC 3100 among the courses taken by the respondent. The available post-survey dataset contains 21 responses; however, the file includes students from multiple courses. To avoid attributing responses from other courses to the COSC 3100 implementation, only the ten post-survey responses that identify COSC 3100 were included in the analysis reported in this chapter. Thus, the descriptive analysis uses $n = 9$ pre-survey responses and $n = 10$ post-survey responses.

An important limitation is that these records cannot be treated as a matched sample of the same students immediately before and after one implementation of the course. The response timestamps span multiple collection periods, and student names were optional. The pre-survey responses include records from January–February 2025 and August 2025, while the COSC 3100 post-survey records include responses from January 2025, December 2025, and April 2026. Consequently, the pre- and post-survey groups are summarized separately. No paired statistical test, effect size, or claim of causal improvement is reported.

Coding of survey responses. Experience questions used a seven-point ordinal response scale. For descriptive analysis, responses were coded numerically from 1 (*No Experience At All*) to 7 (*Extremely Experienced*). Agreement questions were similarly coded from 1 (*Strongly Disagree*) to 7 (*Strongly Agree*). Means are reported to provide compact descriptive summaries, but the distributions and sample sizes should also be considered because the measures are ordinal and the samples are small.

Table 15.4 summarizes selected questions that are particularly relevant to the PDC integration.

Table 15.4: Descriptive summary of selected COSC 3100 survey responses. Pre- and post-survey samples are not matched.

Survey Measure	**Pre** ($n = 9$)	**Post** ($n = 10$)
Problem decomposition experience	4.00	4.10
Parallel-processing experience	2.78	3.00
Distributed-computing experience	2.44	2.60
Interest in computing	5.56	5.80
Interest in learning about PDC	5.33	5.30
Belief that learning PDC is important	5.89	5.80
Perceived understanding of computing	4.89	5.20
Perceived understanding of PDC	4.22	4.20

Prior experience with parallel and distributed computing. The pre-survey results indicate that students entered COSC 3100 with substantially less experience in PDC than with general programming. The mean self-reported parallel-processing experience was 2.78 out of 7. Three of the nine respondents (33.3%) selected *No Experience At All*, and two additional respondents (22.2%) selected *Very Little Experience*. Thus, five of nine students (55.6%) placed themselves in the bottom two categories of the parallel-processing experience scale.

A similar pattern occurred for distributed computing. The mean pre-survey rating was 2.44 out of 7, and five of nine respondents (55.6%) reported either *No Experience At All* or *Very Little Experience*. These results provide useful context for the instructional design: although COSC 3100 students had previous programming experience, substantial prior hands-on experience with PDC could not be assumed.

The COSC 3100 post-survey responses remained heterogeneous. The mean self-reported parallel-processing experience was 3.00 out of 7. Five of ten respondents (50.0%) still selected either *No Experience At All* or *Very Little Experience*. For distributed computing, the post-survey mean was 2.60, and seven of ten respondents (70.0%) selected one of the two lowest experience categories.

These findings are important because they do not support the stronger claim that all students became experienced parallel programmers as a result of the course. Instead, the data suggest that students' self-reported hands-on experience with PDC remained limited and varied considerably among respondents. This is consistent with the relatively lightweight nature of the PDC integration and indicates that future implementations would benefit from additional structured programming practice.

Interest in and perceived importance of PDC. Although students reported relatively limited PDC experience, their attitudes toward learning PDC were generally positive. The mean pre-survey response to interest in learning about parallel and distributed computing was 5.33 out of 7, compared with 5.30 among the COSC 3100 post-survey respondents.

Students also generally considered PDC important. The mean rating for the importance of learning PDC was 5.89 in the pre-survey group and 5.80 in the post-survey group. Eight of nine pre-survey respondents selected one of the positive agreement categories (*Partially Agree*, *Agree*, or *Strongly Agree*). Among the ten COSC 3100 post-survey respondents, nine selected a positive agreement category and one selected *Neutral*.

The combination of relatively high interest and relatively low prior experience is pedagogically significant. It suggests that the principal barrier to introducing PDC at this level was not necessarily student interest; rather, students needed opportunities to translate general interest into concrete programming and performance-analysis experience.

Self-reported understanding. Students were also asked about their perceived understanding of computing and PDC. General perceived understanding of computing averaged 4.89 out of 7 in the pre-survey group and 5.20 in the post-survey group. Perceived understanding of PDC averaged 4.22 in the pre-survey group and 4.20 in the post-survey group.

The similarity of the PDC understanding ratings is particularly important when interpreting the intervention. Because the samples are unmatched, the small numerical difference should not be interpreted as evidence of either improvement or decline. More importantly, the data do not support a claim that students universally reached a high level of PDC understanding after the course. The appropriate conclusion is that respondents generally expressed moderate perceived understanding while reporting relatively limited direct experience with parallel and distributed programming.

General COSC 3100 learning outcomes. The post survey also contained questions addressing broader COSC 3100 learning outcomes. Among COSC 3100 respondents with non-missing answers, students reported relatively high confidence in several traditional CS2 topics. Mean self-ratings were 6.00 out of 7 for applying software-development practices, 6.11 for understanding and applying recursive functions, 5.56 for critically assessing algorithms, 5.78 for describing and using binary trees, and 5.00 for understanding and applying graphs.

These items provide useful context about students' perceptions of the overall CS2 course, but they should not be interpreted as direct measures of the effectiveness of the PDC intervention. In particular, strong self-reported understanding of recursion or software-development practices does not demonstrate mastery of parallel programming.

Assessment limitations. The initial course implementation primarily emphasized exposure to PDC and the feasibility of incorporating parallel-programming examples into an existing CS2 curriculum. PDC-related questions were included in course quizzes; however, item-level quiz results suitable for systematic analysis were not available in the retained data. Therefore, statements such as "most students answered the parallel-programming questions correctly" cannot be quantitatively verified and are not used as evidence of learning gains in this chapter.

Similarly, the available surveys primarily measure experience, attitudes, and self-perceived understanding rather than direct PDC competency. Self-report measures cannot establish whether a student can correctly decompose an algorithm, implement parallel code, identify a synchronization requirement, or interpret parallel performance.

For these reasons, no inferential statistical analysis was conducted. Given the small sample sizes, unmatched observations, different collection periods, and ordinal self-report measures, statistical comparisons of the two group means would imply a level of experimental control that the available data do not provide.

Implications for future assessment. The available data nevertheless provide useful guidance for improving future implementations. A subsequent offering should administer the same assessment immediately before and after the PDC instructional unit and use an anonymous matching identifier so that individual responses can be paired without collecting personally identifiable information. The assessment should also include direct measures of PDC knowledge rather than relying primarily on self-reported experience.

For example, students could be asked to (1) identify which portions of an algorithm can execute concurrently, (2) explain why synchronization is required at a particular point, (3) identify a potential race condition, (4) calculate speedup from measured serial and parallel runtimes, and (5) explain why increasing the number of threads does not necessarily produce proportional performance improvement. A programming-based assessment could additionally require students to execute serial and parallel versions of an algorithm, report repeated timing measurements, and interpret the resulting performance.

Thus, the primary conclusion from the available survey evidence is deliberately limited. COSC 3100 students generally expressed interest in PDC and considered it an important computing topic, while reporting comparatively little direct experience with parallel and distributed computing. The current data do not establish a measurable pre/post learning gain. Instead, they identify both the feasibility of exposing CS2 students to PDC and the need for more systematic hands-on activities and aligned assessment in future implementations.

The following table demonstrates the basic C++ `chrono` timing mechanism used in the instructional material. The timing code provides a foundation for a stronger performance-based assessment because the same measurement procedure can be applied to serial and parallel implementations. Rather than treating one faster execution as proof that parallelization is beneficial, students should repeat the experiment across multiple runs, input sizes, and, where appropriate, thread counts. The resulting measurements can then be used to calculate speedup and to discuss parallel overhead, workload size, and performance variability.

Chapter 16

CS2 course package using Python - Webster University

Python-Based Infusion through Multiprocessing, Parallel Image Processing, and Performance Engineering[†]

Xiaoyuan Suo

Webster University, St.Louis, MO

xiaoyuansuo51@webster.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

16.1 Introduction . . . 444
16.2 How CS2 was changed and PDC topics integrated . . . 444
16.2.1 Course Organization Before and After PDC Integration . . . 444
16.2.2 Highlights . . . 444
16.2.3 Challenges . . . 447
16.3 New Unplugged Activities . . . 448
16.4 New Plugged-in Activities . . . 449
16.4.1 Module 1: Multiprocessing and PDC Basics . . . 449
16.4.2 Module 2: Parallel Image Processing . . . 450
16.4.3 Optional Zombie Attack Assignment . . . 450
16.4.4 Module 3: Performance Engineering . . . 450
16.5 Course-wide Pre and Post Course Surveys for COSC 2110 . . . 452
16.6 Other Activities and Ideas . . . 453
16.7 Conclusion . . . 453
Appendix . . . 454

Chapter figures

16.1 Decomposition of an image into horizontal strips assigned to separate workers. For neighborhood-based operations such as blurring, each strip also requires appropriate neighboring pixel data at internal boundaries. . . . 446
16.2 Distribution of self-reported experience in the pooled post-survey responses ($n = 21$ valid response records per item). All seven experience categories are shown. . . . 456

Chapter tables

16.1 Comparison of course organization before and after PDC integration. . . . 445
16.2 Comparison between parallel programming and vectorization. . . . 447
16.3 Estimated instructional time and student deliverables for the CS2 PDC modules. . . . 448
16.4 PDC concepts matching with performance engineering topics . . . 449
16.5 Illustrative reporting format with invented timings, not classroom measurements. Input: 200,000 integers. Parallel rows use `chunksize=10,000`. Times are seconds. 451
16.6 Descriptive summaries of available pooled survey responses. All items use seven-point coding. Counts refer to valid response records for each item. . . . 452

Abstract

This chapter describes the integration of parallel and distributed computing (PDC) concepts into an existing Python-based CS2 course at Webster University through three complementary modules. The first introduces threading, multiprocessing, task decomposition, and synchronization through demonstrations and a parallel summation assignment. The second uses OpenCV examples of color-based foreground masking and an image-blurring assignment to illustrate image decomposition and distinguish multiprocessing from vectorized array operations. The third introduces performance engineering through sequential and parallel runtime comparisons, repeated measurements, speedup, efficiency, and investigations of process overhead and task granularity. Unplugged activities and programming demonstrations connect these concepts to familiar CS2 topics, including loops, functions, arrays, and file processing. The core activities run on ordinary multicore computers, while HPC demonstrations provide additional context. Descriptive survey results are reported with response counts and limitations; the pooled, unmatched responses do not establish individual learning changes or effects attributable to the modules.

Descriptor Elements

Programming Language Employed: Python

Relevant core courses: CS2, Data Structures, Advanced Python Programming

Prerequisites: CS1, which is taught in C++ at Webster

Textbook: Reference book: Starting Out with Python, 6th edition by Tony Gaddis

Relevant PDC topics:

PDC Concept	Bloom's Level
Introduction to Parallel Computing	Remembering (RE)
Threads and Multiprocessing	Creating (CR)
Race Conditions and Synchronization	Applying (AP)
Task Decomposition and Data Partitioning	Applying (AP)
Performance Measurement and Benchmarking	Analyzing (AN)
Speedup and Efficiency	Understanding (UN)
Strong vs. Weak Scaling	Understanding (UN)
Parallel Overhead	Understanding (UN)
Memory-bound vs. Compute-bound Workloads	Understanding (UN)

Learning outcomes:

1. Explain the differences between sequential, parallel, and distributed computing.
2. Develop basic multithreaded and multiprocessing programs in Python.
3. Identify and resolve race conditions using synchronization mechanisms such as locks and RLocks.

4. Apply task decomposition techniques to parallelize computational workloads.
5. Analyze runtime, speedup, and efficiency of parallel programs using empirical measurements.
6. Interpret scalability behavior using Amdahl's Law and weak/strong scaling concepts.
7. Implement parallel image processing applications using Python multiprocessing and OpenCV.
8. Evaluate the effects of parallel overhead, memory bottlenecks, and workload granularity on performance.

Context for use: The materials were developed for a Python-based CS2 course at Webster University, which is a primarily undergraduate institution (PUI). The course typically enrolls 10-25 undergraduate computer science students with limited prior exposure to parallel computing or computer architecture concepts but with knowledge in C++. The modules are designed to be lightweight and approachable for students without advanced mathematical backgrounds. Python was selected because of its accessibility and extensive ecosystem support for multiprocessing, visualization, and image processing libraries such as OpenCV and NumPy. The activities can be conducted on standard desktop or laptop computers without requiring access to specialized HPC hardware, although optional extensions may leverage multicore servers or institutional clusters.

16.1 Introduction

Python was selected for this course because students enter with prior programming experience in C++. This allows them to focus on computational thinking and PDC concepts rather than low-level syntax. Python also provides a gentle transition into parallel and distributed computing through its simple syntax, rich scientific libraries, and accessibility for STEM students outside traditional computer science backgrounds. In addition, Python is widely used in data science, artificial intelligence, and research computing, which makes it attractive to a larger population of undergraduate students (not only computer science majors).

Despite these advantages, Python is not always ideal for teaching PDC compared to languages such as C++ or Fortran. Python introduces limitations related to the Global Interpreter Lock (GIL), lower execution performance, and reduced exposure to low-level memory management and hardware optimization. As a result, some advanced performance engineering topics must be simplified or demonstrated conceptually rather than implemented at full scale. Nevertheless, Python enables rapid experimentation and visualization, which is particularly valuable in an introductory undergraduate setting.

16.2 How CS2 was changed and PDC topics integrated

Is started integrating PDC topics into this Python course in 2020 through small iterative modifications each year. Introductory PDC concepts are presented in Weeks 4-5, after students have become familiar with core Python data structures such as lists and dictionaries. Module 2, parallel image processing activities are introduced around Week 8 after students gain experience with files and binary data. Module 3, basic performance engineering concepts are introduced near the end of the semester after the students have developed stronger programming and debugging skills.

To make room for the new material, several traditional topics were reduced or streamlined. Less time is spent on low-level image processing details and introductory SQLite programming because these topics are covered more thoroughly in other courses within the curriculum. The revised structure emphasizes computational thinking, scalability, and experimental analysis while preserving the core programming foundations of the original course.

16.2.1 Course Organization Before and After PDC Integration

Table 16.1 summarizes the major differences between the original course structure and the revised syllabus after integrating PDC concepts and activities.

16.2.2 Highlights

Three PDC-related topics are included in the teaching materials.

- **Module 1: Basics of Parallel and Distributed Computing:** This module introduces PDC through everyday workload-sharing examples, followed by Python demonstrations and a parallel summation homework assignment. We start with examples

Table 16.1: Comparison of course organization before and after PDC integration.

Topic Area	**Original Course**	**Revised Course with PDC**
Python fundamentals	Full coverage	Full coverage
Lists and dictionaries	Full coverage	Full coverage with task decomposition examples
File processing	Traditional file I/O	File I/O with parallel image processing examples
Image processing	Detailed sequential coverage (such as OpenCV and Pillow),	Reduced depth; integrated parallel image processing activities such as image blurring and black-and-white conversion.
SQLite / Python database programming	Multiple weeks	Reduced to 1 session, as SQLite is covered in other classes
Introduction to PDC	Not included	Added in Weeks 4–5
Parallel image processing	Not included	Added in Weeks 8–9
Performance engineering	Not included	Added near end of semester
Experimental benchmarking	Minimal emphasis	Added through timing and scalability exercises

involving shoveling snow, digging holes, and planting trees, prompting students to reason about task decomposition, dependencies, and how the number of workers affects completion time. Parallel and distributed computing are introduced conceptually, while the programming examples focus on threads and processes within a single computer. The demonstrations begin with creating threads, identifying the thread that is currently running, and defining a thread through subclassing. Subsequent examples introduce concurrent access to shared data, race conditions, and synchronization using `Lock` and `RLock`. The slides explain lock acquisition and release and illustrate deadlock through two threads that each hold a lock needed by the other. These examples connect execution order and resource dependencies to program correctness. my slides The module then introduces multiprocessing, including process creation and the use of `multiprocessing.Pool` to apply a function to multiple inputs. Examples contrast blocking calls with asynchronous task submission and result retrieval. The accompanying sample-program collection includes thread-management, race-condition, locking, multiprocessing, process-pool, and minimum-finding examples. The summation homework extends these demonstrations to a larger numerical workload, providing a concrete setting for applying decomposition and combining partial results.

- **Module 2: Parallel Image Processing, Multiprocessing, and Vectorization:** I introduced parallel image processing through visual demonstrations and a homework assignment using Python and OpenCV. Students subsequently parallelized a supplied Gaussian-blur program, applying decomposition while considering neighboring pixels at partition boundaries. Images provide a concrete representation of data decom-

position: students can see how a larger input is divided into regions, processed by different workers, and reassembled (Figure 16.1). The examples connect familiar array operations and function calls to parallel computation without requiring students to implement image-processing algorithms from scratch.

I also show a worked demonstration that removes a green background from an image. The initial program converts the image to HSV color space, identifies green pixels using a color-range mask, and retains the foreground. The multiprocessing version divides the image into horizontal strips, applies the same processing function to each strip through a process pool, and combines the results using `NumPy.vstack`. Presenting the two implementations together makes the decomposition and result-collection steps explicit. The before-and-after images provide a visible illustration of the transformation.

The demonstration also distinguishes multiprocessing from vectorized library operations (Table 16.2). The initial implementation already uses OpenCV operations on image arrays rather than explicit pixel-by-pixel Python loops. The multiprocessing version adds a separate level of concurrency by distributing image regions among worker processes. This comparison shows that array operations and multiprocessing can be used together, while raising the question of whether the additional process-management and data-transfer costs are justified for a given image size. As a follow-up homework assignment, students parallelize a supplied Gaussian-blur program. This extends the decomposition exercise to an operation that uses neighboring pixels, making the treatment of partition boundaries an important correctness consideration.

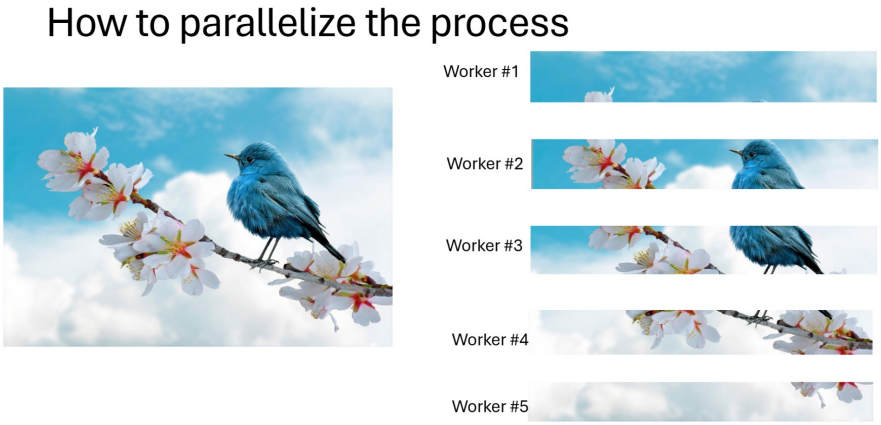


Figure 16.1: Decomposition of an image into horizontal strips assigned to separate workers. For neighborhood-based operations such as blurring, each strip also requires appropriate neighboring pixel data at internal boundaries.

- **Module 3: Introduction to Performance Engineering:** This module introduces performance engineering through short Python experiments on an ordinary multicore computer. The teaching materials emphasize measuring a sequential baseline, repeating timing trials, and interpreting runtime, speedup, and efficiency, mostly at the conceptual level. Worked examples use `time.perf_counter()` and the median of three runs to illustrate why a single measurement is insufficient. Multiprocessing

examples examine process creation, task scheduling, data serialization, and communication overhead. In one example, students can compare a sequential loop that squares 200,000 integers with a process-pool implementation and investigate how changing `chunksize` affects runtime while preserving the computation. Additional examples vary process counts and illustrate the costs of transferring large inputs. These experiments provide concrete starting points for discussing why additional processes may produce limited speedup or longer runtimes. The slides also introduce scaling laws, compute-bound versus memory-bound workloads, and performance plots as tools for interpreting observations.

The CS2 materials are available in Module 3: Performance Engineering. The separate upper-level performance engineering collection is an optional extension. It is not a prerequisite for the CS2 module and does not add requirements to the CS2 activities. NUMA investigations and other hardware-specific extensions are optional and are outside the required CS2 experiment.

Table 16.2: Comparison between parallel programming and vectorization.

Feature	**Parallel Programming (Multiprocessing)**	**Vectorization (Array Operations)**
Level	Multi-core execution (high-level parallelism)	Array operations in compiled libraries; may use SIMD or internal threads
Real-world analogy that I used in class	Hiring four chefs to cook four different pizzas simultaneously	Using a specialized machine to cook four pizzas at once, or a large stamp-like cutter to slice many pepperoni pieces simultaneously
Common Python tools	`multiprocessing`, `joblib`	`NumPy`, `Pandas`
Hardware focus	Multiple CPU cores	SIMD and memory access; library-dependent use of CPU cores

Student Deliverables and Estimated Time for Each Module The instructional times are approximate and can be adjusted according to the existing CS2 schedule and students' prior programming experience. The three primary modules require approximately three 50-minute class periods in total. The Zombie Outbreak activity serves as a more substantial programming assignment that reinforces the same concepts through independent practice; its placement in either CS1 or CS2 depends on which course I am teaching during a given academic cycle. See table 16.3. The estimates refer to classroom instruction; programming completion, repeated measurements, and written analysis may require additional time outside class.

16.2.3 Challenges

Limitations of Using Python for PDC One challenge of teaching PDC in Python is that the language is not designed primarily for high-performance computing. Compared to C++ or Fortran, Python provides less direct control over memory management, thread scheduling, and hardware-level optimization. The Global Interpreter Lock (GIL) also limits its true multithreaded execution in many situations, requiring instructors to simplify some discussions or rely on higher-level libraries and multiprocessing approaches.

Table 16.3: Estimated instructional time and student deliverables for the CS2 PDC modules.

Module/Activity	Estimated Time	Student Deliverables
Basics of Parallel and Distributed Computing	1 class period (50 min.)	Completion of short in-class Python exercises demonstrating task decomposition, multiprocessing, synchronization, and basic concurrency concepts. Students answer brief conceptual questions about race conditions, locks, deadlocks, and the differences among sequential, parallel, and distributed execution.
Parallel Image Processing, Multiprocessing, and Vectorization	1 class period (50 min.)	Completion of an image-processing exercise (available in the slides) in which students decompose an image-processing problem and compare alternative implementations. Students examine multiprocessing and vectorized approaches and provide a short interpretation explaining how the approaches differ and why their performance may differ.
Introduction to Performance Engineering	1 class period (50 min.)	A small performance experiment in which students measure program runtime, compare execution configurations, and calculate or interpret basic performance measures such as speedup and efficiency. Students submit timing results together with a brief explanation of observed performance trends and possible bottlenecks.
Zombie Outbreak Parallelization Homework (optional/conditional)	1–2 hours outside class	Students modify and experiment with a "Zombie Outbreak" simulation to apply task decomposition and parallel programming concepts in a larger programming problem. They compare sequential and parallel execution and briefly discuss correctness and performance. This assignment is used in CS2 when I am not teaching CS1; when I teach CS1, the activity is assigned in CS1 instead to avoid unnecessary duplication across the curriculum. When used in CS2, the difficulty is slightly higher, I add randomly generated Zombies, substantially increase the grid size and ask students to focus on performance rather than completing the program.

Despite these limitations, Python is still appropriate for introducing PDC at the CS2 level because its accessible syntax and high-level parallel programming libraries allow students to focus on fundamental concepts such as task decomposition, concurrency, communication, and performance measurement without being overwhelmed by low-level implementation details.

The Lack of an Appropriate Textbook Another challenge is the lack of beginner-friendly textbooks that integrate Python programming with introductory PDC concepts. Most available PDC materials assume stronger mathematical or systems programming backgrounds and are commonly written using C/C++ with MPI or OpenMP. Therefore, I used considerable effort to develop and custom teaching materials, animations, and assignments, making them suitable for early undergraduate students.

Generative AI As mentioned in the previous chapter, modern generative AI tools create challenges in assignment design and assessment. Many introductory programming exercises can now be answered quickly using AI-assisted coding tools. As a result, my assignments now increasingly emphasize experimentation, interpretation of results, performance analysis, and reflection rather than code implementation alone.

16.3 New Unplugged Activities

The new unplugged activity is the dispatcher-and-workers role-play described below. It builds on the conceptual activities in the Webster CS1 chapter. As a separate plugged-in demonstration, we used an interactive linked-list animation to illustrate parallel access to

a shared data structure. The animation allows students to visualize multiple operations occurring concurrently on a linked list and provides a concrete transition from familiar CS2 data structures to concepts such as concurrent access, synchronization, and race conditions. The animation is available at Parallel Linked-List Access Animation.

For distributed computing, I developed a simulation activity in which students role-play as users (3), a dispatcher node (1), and worker processors (other students). Users submit some simple computational tasks to a central dispatcher, which distributes the work among available processors and collects the results. Worker processors may enter states such as *idle*, *running*, *waiting*, or *failed*, allowing students to visualize concepts including task scheduling, communication overhead, load balancing, synchronization, scalability, and fault tolerance in distributed systems.

Table 16.4 summarizes the concepts demonstrated by this activity.

Table 16.4: PDC concepts matching with performance engineering topics

PDC Concept	**How the Activity Demonstrates It**
Task distribution	Dispatcher assigns jobs to workers
Load balancing	Some workers may become overloaded while others remain idle
Communication overhead	Time spent passing messages between users, dispatcher, and workers
Fault tolerance	A worker may "fail" before completing a task
Scalability	More workers can process more jobs simultaneously
Synchronization	Results must return before final answers are assembled
Resource management	Dispatcher decides which worker receives tasks
Distributed systems architecture	Multiple independent processors collaborate on one overall problem

16.4 New Plugged-in Activities

16.4.1 Module 1: Multiprocessing and PDC Basics

Students complete the short Python exercises described in the module highlights and deliverables table, exploring task decomposition, process pools, synchronization, and race conditions. The dispatcher-and-workers activity supplies a conceptual introduction.

Programming assignment on Summation. The module concludes with a homework assignment requiring students to implement a parallel version of `find_sum`. The assignment builds on familiar loops and accumulation operations. The supplied problem specifies randomly generated numerical data organized into five rows, with 10,000,000 values per row, for a total of 50,000,000 values. Students must determine how to divide the summation

workload among processes and combine the partial sums into the final result. The assignment connects familiar iteration and accumulation operations to parallel decomposition and reduction. I require students to do most of the discussions and algorithm designs in a lab session to reduce the usage of AI.

16.4.2 Module 2: Parallel Image Processing

In class, I demonstrated green-background removal using OpenCV, then showed how the image could be divided into horizontal strips and processed through a process pool. Students partition an image-processing workload, compare sequential, multiprocessing, and vectorized implementations, and explain correctness and runtime observations. The image-processing figure and module highlights describe the accompanying visual examples.

16.4.3 Optional Zombie Attack Assignment

A description of the Zombie Attack assignment appears in the Webster CS1 chapter. The Python version introduces random zombie placement and slightly increases the difficulty. It also places greater emphasis on performance analysis, requiring students to evaluate and explain their program's behavior rather than simply submit AI-generated code. The Python Zombie assignment is available Zombie Homework.

16.4.4 Module 3: Performance Engineering

Performance engineering was introduced as a new plug-in activity to connect parallel programming with the practical question of whether a program actually runs faster. Using HPC-based demonstrations and short Python examples, I showed how execution time changes as computational resources and workload sizes vary. Students examined sequential and parallel runtimes and used speedup and efficiency to discuss whether additional processing resources were being used effectively. Examples involving process-pool overhead and task size provided opportunities to examine why adding more processes does not always improve performance.

I devoted approximately one 50-minute class session, with an optional second session, to demonstrating measurements and discussing the results. The slides emphasize establishing a sequential baseline, repeating timing trials, and interpreting performance trends. HPC demonstrations provided a larger-scale context, while the core exercises can also run on an ordinary multicore computer. Detailed slides and examples are available in the CS2 Performance Engineering Teaching Module. Separate upper-level performance engineering materials are optional extensions and are not required for this CS2 activity.

Performance Experiment A focused experiment can use the integer-squaring example from the slides to investigate task granularity and multiprocessing overhead. The protocol below makes the inputs, comparisons, and reporting requirements explicit. It can be adopted as a guided exercise or a short homework extension.

1. **Input and baseline.** Use `range(0-200,000)` as the input and compute `x * x` for each value. The sequential baseline constructs the complete output list using the same worker function as the parallel version. Keep the input size fixed across configurations.
2. **Process counts and chunk sizes.** Compare the sequential baseline with process pools containing 1, 2, and 4 workers where the available hardware permits. First vary `chunksize` through 1, 100, and 10,000 with a fixed pool size. Then vary the worker count with a fixed chunk size. Changing one parameter at a time helps separate the comparisons. Record the machine's available CPU count and any omitted configurations.
3. **Repeated trials.** Perform one untimed warm-up for each configuration, followed by three measured trials. Record every trial and report the median. Use the same machine and software environment throughout, and record the Python version and operating system.
4. **Timing boundaries.** Create the input before starting the timer. For the sequential baseline, time the computation and output-list construction. For each parallel trial, start the timer before creating the pool and stop it after collecting the complete output and exiting the pool context. Thus the parallel measurement includes pool startup, scheduling, data transfer, and shutdown. Exclude printing, plotting, and correctness checks from both timed regions. Create a new pool for each measured trial to keep this end-to-end comparison consistent.
5. **Correctness.** Compare each parallel output list with the sequential output list outside the timed region. All integer results and their ordering must agree before interpreting performance.
6. **Performance measures.** Let T_{seq} be the median sequential runtime and T_p the median runtime with p worker processes. Calculate $S_p = T_{\text{seq}}/T_p$ and $E_p = S_p/p$. The one-worker pool is a separate overhead comparison; it does not replace the sequential baseline. Here p denotes worker processes rather than a verified count of exclusively assigned physical cores.
7. **Student submission.** Submit the code, environment information, correctness-check results, all trial times, a summary table, and a runtime plot. Explain how chunk size and worker count affected the observations. A slowdown is a valid result. Distinguish measured behavior from possible explanations, since runtime alone does not identify a particular bottleneck.

Table 16.5 shows a student reporting format.

Table 16.5: Illustrative reporting format with invented timings, not classroom measurements. Input: 200,000 integers. Parallel rows use `chunksize=10,000`. Times are seconds.

Configuration	Trial 1	Trial 2	Trial 3	Median	S_p	E_p
Sequential	0.020	0.019	0.021	0.020	1.00	—
Pool, $p = 1$	0.110	0.120	0.115	0.115	0.17	0.17
Pool, $p = 2$	0.100	0.105	0.110	0.105	0.19	0.10
Pool, $p = 4$	0.120	0.130	0.125	0.125	0.16	0.04

The invented values illustrate how to report a parallel slowdown. Actual timings must

come from students' own runs. Because each integer-squaring task performs very little computation, this example is intended to investigate overhead; increasing chunk size does not guarantee that multiprocessing will outperform the sequential version. Speedup and efficiency in the table are rounded from the median times.

16.5 Course-wide Pre and Post Course Surveys for COSC 2110

The Python-based COSC 2110 course is offered once per year, with typical enrollment of approximately 15–25 students per offering. PDC integration began in early 2020. These course-context figures are separate from survey response counts. The available pooled instructional survey exports contain 10 pre-survey response records and 21 post-survey response records, all indicating consent. These totals describe submitted records, not verified counts of unique students or respondents from a single COSC 2110 offering. Consequently, course-specific response rates cannot be calculated from these exports. The two sets of responses were analyzed as unmatched groups, without linking individuals across administrations. Optional identifying fields were excluded from the analysis.

Table 16.6 reports descriptive summaries for six common items. Each item has 10 valid pre-survey responses and 21 valid post-survey responses. Experience responses were coded on a seven-point scale: 1 = no experience, 2 = very little experience, 3 = slightly experienced, 4 = between slightly and moderately experienced, 5 = moderately experienced, 6 = very experienced, and 7 = extremely experienced. Agreement responses were coded from 1 = strongly disagree to 7 = strongly agree, with partially disagree, neutral, and partially agree as intermediate categories. Means summarize these ordinal ratings and should be interpreted alongside the response categories.

Table 16.6: Descriptive summaries of available pooled survey responses. All items use seven-point coding. Counts refer to valid response records for each item.

Item	Pre n	Pre M	Post n	Post M
Problem decomposition experience	10	4.10	21	3.52
Parallel processing experience	10	2.60	21	2.19
Distributed computing experience	10	2.30	21	2.05
Interest in learning about PDC	10	5.50	21	4.76
Perceived importance of learning PDC	10	5.70	21	5.24
Self-reported understanding of PDC	10	4.10	21	3.71

Mean ratings for interest in learning about PDC and its perceived importance were above the neutral midpoint of 4 in both sets of responses. Parallel and distributed computing experience ratings were lower than problem decomposition experience ratings in both sets. Post-survey means were lower than pre-survey means for all six items. These differences describe the available response groups and do not establish changes within individual students, learning gains or losses, or effects attributable to the Python modules. Differences in respondent composition and survey participation limit interpretation. The responses measure reported experience and perceptions rather than demonstrated programming skill.

16.6 Other Activities and Ideas

CS2 vs. CS1 My experiences from multiple years of this course offering suggest that these PDC topics are difficult to integrate effectively into an introductory Python programming course. Many concepts, including task decomposition, scalability, benchmarking, and multiprocessing, require students to already possess sufficient programming maturity and debugging skills. As a result, the above-mentioned material is better suited for a CS2-level course, where students already have prior exposure to programming fundamentals and basic data structures.

HPC Topics Future versions of the course may incorporate additional HPC-related activities, including introductory cluster usage, remote job submission, and simple distributed computing demonstrations using shared computing resources. Introducing students to HPC systems early may help bridge the gap between traditional programming instruction and modern computational research practices.

16.7 Conclusion

This chapter describes the integration of PDC into an existing Python-based CS2 course through three modules: parallel programming basics, parallel image processing, and performance engineering. The modules combine conceptual demonstrations with programming activities. They build on existing course topics such as loops, functions, arrays, and file processing, while selected image-processing and database topics were streamlined to accommodate the new material.

Performance experiments compare sequential execution, multiprocessing, and array-based library operations, with attention to runtime, speedup, efficiency, and overhead. The core activities use ordinary multicore computers; HPC demonstrations provide additional context, and upper-level extensions remain optional. The survey results describe respondents' reported experience and perceptions. Because the responses are pooled and unmatched, they do not establish individual learning changes or effects attributable to the modules.

Appendix

GitHub Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter16-Webster

Detailed Course Organization Before and After PDC Integration

The course syllabus is linked at COSC 2110 Spring 2026 Course Syllabus.

The course follows a traditional Python programming sequence during the first half of the semester. Weeks 1–3 focus on Python syntax, control structures, functions, and file processing. During this period, students practice reasoning about computational tasks and data organization.

PDC concepts are introduced during Weeks 4–5, after instruction on lists, dictionaries, and sets. Rather than introducing formal parallel algorithms immediately, Module 1 emphasizes task decomposition, workload distribution, multiprocessing, synchronization, race conditions, and deadlocks. These topics are reinforced using unplugged activities, classroom demonstrations, and simple Python multiprocessing examples. In the instructor's experience, students need practice with basic Python syntax and debugging before attempting these activities; this observation is not a measured instructional effect.

Module 2 introduces parallel image processing around Weeks 8–9, after recursion and file processing. Image processing provides a visual and naturally parallel problem domain. Students experiment with grayscale conversion, image decomposition, and filtering while discussing data partitioning and vectorization. NumPy array operations may use SIMD instructions or internally threaded libraries; vectorization does not itself imply multiple Python processes or exclusively single-core execution.

Module 3 introduces basic performance engineering near the end of the semester. Activities focus on benchmarking, runtime measurement, speedup, efficiency, scalability, and multiprocessing overhead. Students document runtime behavior, compare sequential and parallel implementations, and analyze cases where additional processes do not reduce runtime because of overhead or other bottlenecks.

The three core modules each have an estimated duration of one 50-minute class period. A second performance engineering session and the Zombie Outbreak homework are optional extensions. Ordinary multicore laptops or desktops are sufficient for all three core modules. Large-cluster experiments, deep-learning workloads, and hands-on NUMA investigations are optional advanced extensions.

Organization of Material

The instructional package is organized into three modules containing lecture slides, assignments or experimental exercises, instructor notes, and supporting documents. The inventory below describes the materials for each module. The chapter's CDER repository is linked above.

1. **Module 1: Multiprocessing and PDC Basics.** The module materials include `Python parallel.pdf`, `Assignment12`, and `Read me.pdf`.
   Introduces PDC through Python examples and real-world analogies. Topics include multiprocessing, synchronization, locks, race conditions, deadlocks, and task decomposition. The programming assignment asks students to implement parallel summation and document runtime behavior. Students complete a substantial portion of the implementation during supervised class activities and document observations independently. Instructor notes provide suggestions for demonstrations, pacing, and discussions.
2. **Module 2: Parallel Image Processing and Vectorization.** The module materials include `parallel image.pdf`, `Assignment`, and `read me.pdf`.
   Introduces image blurring, grayscale conversion, and image decomposition. Students compare multiprocessing with vectorized NumPy operations to distinguish concurrent process execution from array-based computation. Instructor notes address conceptual discussions and implementation details.
3. **Module 3: Performance Engineering.** The module materials include `Basic Performance Engineering Python.pdf`, `Performance engineering notes.pdf`, and `read me.pdf`.
   Introduces benchmarking, runtime measurement, scalability, speedup, efficiency, multiprocessing overhead, and experimental analysis. Supplemental notes provide concise explanations and examples. Exercises emphasize documenting and interpreting experimental results alongside program correctness. Instructor notes include implementation suggestions and classroom discussion points.

Survey and Other Anonymized Data and Analysis

The survey analysis uses the supplied pre- and post-survey CSV exports, containing 10 and 21 response records, respectively. All records indicate consent to participate. These are pooled instructional survey responses and should not be interpreted as verified respondent counts for a single COSC 2110 offering. Typical course enrollment of approximately 15–25 students per offering is separate from these survey counts. No individual matching or duplicate removal was performed, so the record totals are not established counts of unique participants. Optional identifying fields were excluded from the analysis and are not reproduced here.

Six common items were selected for comparison: experience with problem decomposition, parallel processing, and distributed computing; interest in learning about PDC; perceived importance of learning PDC; and self-reported understanding of PDC. Each selected item has 10 valid pre-survey responses and 21 valid post-survey responses. Experience categories were coded from 1 to 7 in the following order: no experience, very little experience, slightly experienced, between slightly and moderately experienced, moderately experienced, very experienced, and extremely experienced. Agreement categories were coded from 1 = strongly disagree to 7 = strongly agree, with partially disagree, neutral, and partially agree as intermediate categories. Minor capitalization and spelling variations in category labels were treated as equivalent. Item means were calculated as the sum of coded ratings divided by

the number of valid responses and rounded to two decimal places. These means summarize ordinal response categories.

Figure 16.2 shows the distribution of post-survey experience ratings for problem decomposition, parallel processing, distributed computing, and event handling. The bars represent response counts from the pooled post-survey data ($n = 21$ per item).

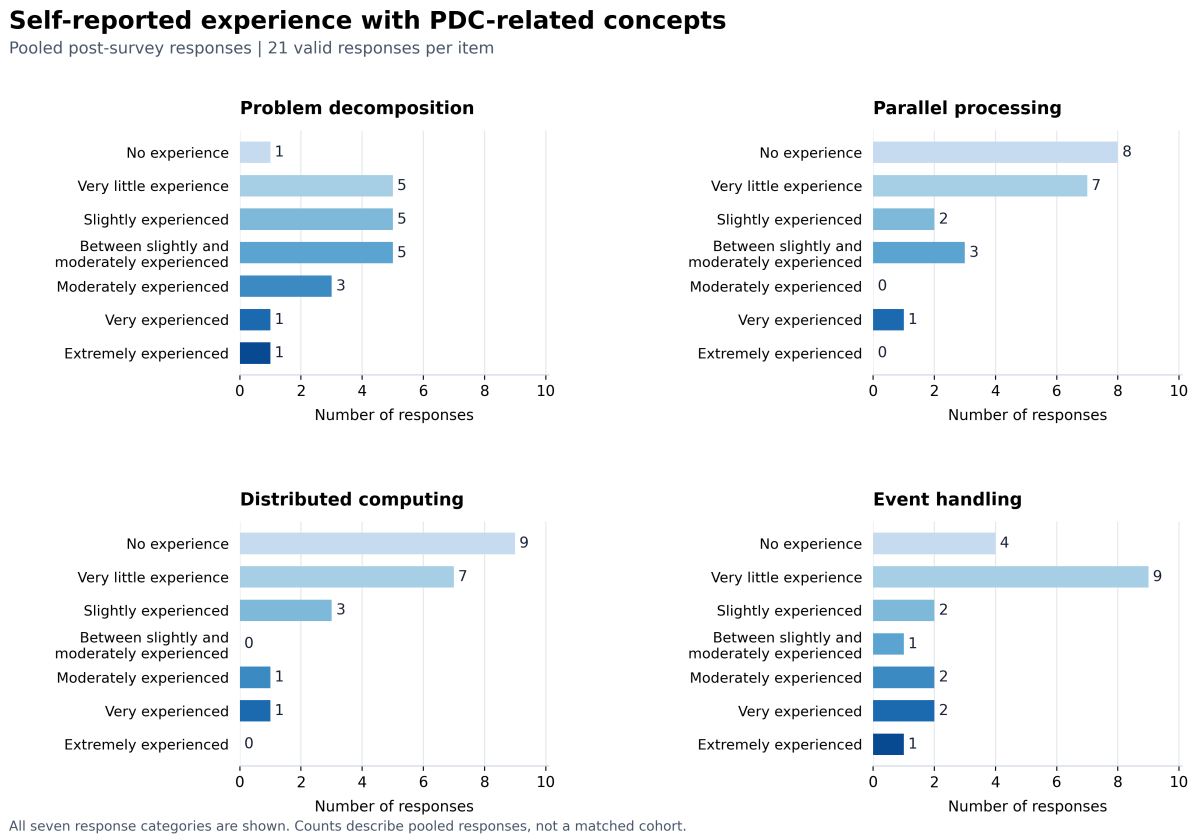


Figure 16.2: Distribution of self-reported experience in the pooled post-survey responses ($n = 21$ valid response records per item). All seven experience categories are shown.

Pre- and post-survey responses are compared as unmatched groups. Differences between their means do not establish within-student changes, learning gains or losses, or causal effects of the Python modules. Course-specific response rates cannot be established from these pooled exports.

Chapter 17

CS2 Course Package – Casper College

Community College Adoption across Unplugged, OpenMP, and Greenfoot Activities[†]

Charlotte Gruner

Casper College, Casper WY

charlotte.gruner@caspercollege.edu

[†]**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

17.1 Introduction . . . 462
17.2 How CS2 was changed and PDC topics integrated . . . 462
17.2.1 Integrated Syllabi (Pre and Post) . . . 462
17.2.2 Highlights . . . 462
17.2.3 Challenges . . . 463
17.3 New Unplugged Activities . . . 463
17.3.1 Unplugged Penny Activity . . . 463
17.3.1.1 Implementation Details . . . 464
17.3.1.2 Experience and Teaching Tips . . . 464
17.3.2 Unplugged Coin Addition . . . 464
17.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 464
17.3.2.2 Implementation Details . . . 465
17.3.2.3 Experience and Teaching Tips . . . 466
17.3.3 Fork-Join Activity . . . 466
17.3.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 466
17.3.3.2 Implementation Details . . . 467
17.3.3.3 Experience and Teaching Tips . . . 467
17.4 New Plugged-in Activities . . . 468
17.4.1 Plugged-In OpenMP Array Creation Activity . . . 468
17.4.1.1 Implementation Details . . . 468
17.4.1.2 Experience and Teaching Tips . . . 469
17.4.1.3 Data and Analysis . . . 469
17.4.2 Plugged-In OpenMP Array Sum Activity . . . 469
17.4.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 469
17.4.2.2 Implementation Details . . . 470
17.4.2.3 Experience and Teaching Tips . . . 470
17.4.2.4 Data and Analysis . . . 471
17.4.3 Greenfoot Activity . . . 471
17.4.3.1 Problem Description, Prerequisites, and Learning Outcomes . . . 471
17.4.3.2 Implementation Details . . . 472
17.4.3.3 Experience and Teaching Tips . . . 472
17.4.3.4 Data and Analysis . . . 472
17.4.4 Plugged-In Sorting Algorithm Comparison . . . 473
17.4.4.1 Problem Description, Prerequisites, and Learning Outcomes . . . 473
17.4.4.2 Implementation Details . . . 473
17.4.4.3 Experience and Teaching Tips . . . 473
17.4.4.4 Data and Analysis . . . 474
17.5 Course-wide Pre- and Post- Course Surveys . . . 474
17.6 Conclusion . . . 478
Appendix . . . 480

Chapter figures

17.1 CASPER CS2 Example of Fork-Join Materials. . . . 467
17.2 CASPER CS2 Example of Fork-Join Solution for problem in Figure 17.1. . . . 467
17.3 CASPER CS2 Part 3 of Plugged-In OpenMP Array Sum Activity . . . 471
17.4 CASPER CS2 Sample Survey Question Reflecting on Level of Experience . . . 474

17.5 CASPER CS2 Sample Survey Question Reflecting on Level of Interest in Computing and PDC 477

Chapter tables

17.1 CASPER CS2 Relevant PDC Topics and Bloom's Levels 461
17.2 CASPER CS2 Baseline Fall 2024 versus Fall 2025 PDC Intervention. PDC Interventions are highlighted in yellow. 463
17.3 CASPER CS2 Survey Results for Experience Questions 475
17.4 CASPER CS2 Survey Results Experience with PDC Topics 476
17.5 Survey Results for CS2 Interest Questions- Casper College 477
17.6 CASPER CS2 Survey Results Interest Questions 477
17.7 CASPER CS2 Survey Results for Understanding Questions 478

Abstract

This chapter presents the integration of Parallel and Distributed Computing (PDC) concepts into an introductory computer science course (CS2) at Casper College, a two-year public institution. The primary goal was to embed foundational PDC topics into the existing curriculum without displacing traditional core content such as classes and data structures. The pedagogical approach utilized a combination of unplugged activities and plugged-in programming assignments in primarily C++ with Java/Greenfoot. Preliminary analysis of survey data collected in a baseline semester in which no PDC interventions were implemented and a semester in which several of the described interventions were implemented indicate that student perception of learning gains in non-PDC related topics remained generally the same for some topics, but were reduced in other topics. Student perception of learning gains in PDC topics was notably higher, in general, in the semester in which PDC interventions were implemented.

Descriptor Elements

Programming Language Employed: primarily C++, Java (for Greenfoot project)

Textbook used: REVEL edition of Starting out with C++ by Tony Gaddis[22] (Spring 2026)

Relevant core courses: CS2 (COSC 2030)

Pre-requisites: CS1 (COSC 1030)

Relevant PDC topics:

Relevant PDC topics and Blooms levels Table 7.1 lists the integrated PDC topics with Bloom's classification (Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR)).

Learning outcomes:

1. Define and recognize fundamental PDC terms such as multi-core, data parallelization, and task parallelization.
2. Apply OpenMP directives to parallelize array processing and analyze the resulting performance gains through benchmarking.
3. Identify and explain race conditions and shared resource conditions in both physical simulations and C++ code.
4. Create a novel interactive 2-D game using the Greenfoot API to demonstrate mastery of event-driven programming and object-oriented design.
5. Demonstrate knowledge of fork-join terminology and be able to analyze a dependency graph to determine best and worst case scenarios.

Context for use: Casper College is a 2-year public college located in central Wyoming. The total enrollment at Casper College is typically between 4500 students and 5000 students. The most popular programs are in general studies, health care, and agriculture

Table 17.1: CASPER CS2 Relevant PDC Topics and Bloom's Levels

PDC Concept	Chapter Section						
	17.3.1 Unplugged Penny	17.4.1 Plugged-In Array Create	17.3.2 Unplugged Coin Addition	17.4.2 Plugged-In Array Sum	17.3.3 Fork-Join Activity	17.4.3 Greenfoot	17.4.4 Plugged-in sort
PDC Vocabulary and Foundations	RE,UN				RE,UN		
Data Parallelism	RE	AP,AN	UN	AP,AN			AP
Race Condition Synchronization			UN	UN,AN			
Speedup and Benchmarking		UN,AN		AN			AN
Distributed Computing							
Event Driven Programming						AP	
Task Parallelism	UN				AP,AN		

fields. The Computer Science Department, which is housed within the Mathematics Department in the School of Science generally consists of around 50 declared majors with 80% working on freshman level coursework and 20% working on sophomore level courses. Demographically, Casper College has more students who are older than the typical 18-21 age group found at a typical university in the United States and less than half of the students are full time. Casper College also attracts local high school students who have the opportunity to take College Classes for both high school and college credit both at their local high schools through concurrent enrollment and on the Casper College campus through dual enrollment. The Computer Science Department has one full time faculty member. The Computer Science Associates Degree at Casper College is meant to serve as the first two years of a 4-year degree program and includes the following core computer science classes: CS0 (COSC 1010) Introduction to Computer Science, CS1 (COSC 1030) Computer Science I, CS2 (COSC 2030) Computer Science II, (COSC 2150) Computer Organization, (COSC 2300) Discrete Structures. Class sizes for 1000-level courses tend to be below 20. Class sizes for 2000-level courses tend to be under 10. At Casper College the CS2 course, COSC 2030 is classified as a 3 hour lecture, 2 hour lab class, but it is currently taught in two weekly class meetings of 135 minutes each. The CS2 class is currently only offered in the fall semester.

17.1 Introduction

While it is generally recognized that introducing Parallel Distributed Computing (PDC) content to computer science students earlier and with more regularity particularly in the first two years of the computer science core classes would be advantageous, how to do this without sacrificing coverage of traditional topics is not clear. This chapter discusses the attempt to include curricular elements adapted from the Development Teams' curricula to introduce PDC topics into the traditional Casper College CS2 in the Fall of 2025 course in a meaningful way for students without sacrificing core CS2 topics. To meet this objective, we focused on unplugged activities and some introductory coding exercises. Much of the PDC material was completed during class time. To make room for the in-class PDC materials some class time normally spent on additional examples and time for independent student work on the traditional CS2 topics was shortened. These activities were shifted to video and/or reading content to be completed outside of class time. The major exception to this approach was the inclusion of the Greenfoot project materials, which was added as a midterm project for the students that was largely self-directed and done outside of class. In this case, we felt that students would be able to watch the tutorial videos for this activity and make progress on their own.

17.2 How CS2 was changed and PDC topics integrated

We introduced some aspect of PDC every week for weeks 1-5 in either an in class unplugged activity, in class plugged in activity, or a short homework assignment. We felt that the PDC material would have a more long term impact if delivered in frequent (weekly), small units rather than only once or twice during the semester. During these weeks we were also covering the same material that is normally covered in the class with only minimal adjustments. These first 5 weeks included three short unplugged activities and an introduction to OpenMP. Because of the timing of this course during the first semester of PDC intervention at Casper College, this cohort of CS2 students had not had any PDC intervention in CS1. Therefore, we chose to include many of the interventions that were originally designed for CS1. In weeks 6-9, students were assigned a midterm project using Greenfoot. This serves as an introduction to event handling, a core PDC topic. In the second half of the semester, the students did one more in-class activity using OpenMP to parallelize a version of Quicksort during a discussion of sorting algorithms.

17.2.1 Integrated Syllabi (Pre and Post)

A summary of the Fall 2025 PDC material added and the timing of the material relative to the baseline CS2 material is shown in Table 17.2.

17.2.2 Highlights

The highest impact change to the syllabus from Fall 2024 to Fall 2025 was the inclusion of the Greenfoot-based midterm project. Prior to Fall 2025, there was no opportunity for

Table 17.2: CASPER CS2 Baseline Fall 2024 versus Fall 2025 PDC Intervention. PDC Interventions are highlighted in yellow.

Week	Pre (Baseline Fall 2024)	Post (PDC content added Fall 2025)
1	Review pointers/classes	Unplugged Penny Activity
2	C-strings, string class	Plugged-In OpenMP Array Creation Activity
3	Classes (instance, static members, friends, member-wise assignment, copy constructors	Plugged-In Array Summation Part 1
4	Classes (operator overloading)	Unplugged Coin Addition Activity
5	Classes (object conversion, Aggregation, Rvalue References and Move Semantics)	Plugged-In OpenMP Array Summation completion, fork join activity
6	Classes (inheritance)	Intro to Greenfoot
7	Classes (polymorphism, abstract base class)	Greenfoot project
8	Templates	Greenfoot project
9	STL Library	Greenfoot project presentations
10	STL	—
11	Linked Lists	—
12	Stacks & Queues	—
13	Recursion	—
14	Binary Trees	Plugged-In Sorting Algorithm Comparison
15	Searching, Sorting, Big O, AVL Trees, Heaps, Hashes	no new content

independent project work, which was clearly of interest to the students. The Greenfoot project filled this gap in addition to providing students with an experience with event handling. Importantly, both the intervention and non-intervention courses covered the same core CS2 topics. It is also important to emphasize that many of the interventions used in the Fall 2025 intervention semester were designed for CS1 and were used with this cohort because they had not had any previous interventions in CS1. The long term curriculum will assume students have encountered the unplugged basics in CS1, for example Unplugged Penny Activity.

17.2.3 Challenges

The greatest pedagogical hurdle with the modules added was the C++ coding exercises using `OpenMP`. Even greatly simplified, the use of `OpenMP` and the `chrono` libraries has high intrinsic cognitive load and requires significant time for students to understand how to use these libraries. We did find that this material seems to be easier for CS2 students to take on these challenges more successfully and more independently than CS1 students. Also, since most of the `OpenMP` material was presented early in the semester, we found that in week 14 of the semester when we returned to this material to do one more `OpenMP` activity (Plugged-In Sorting Algorithm Comparison, Section 17.4.4), the students required review of this material.

17.3 New Unplugged Activities

17.3.1 Unplugged Penny Activity

Prerequisites No prerequisites are required for this activity.

Learning Outcomes

Students will be able to:

- explain the difference between sequential and parallel processing.
- describe how synchronization steps ensure a final, correct output when multiple workers are involved.
- recognize instances where the overhead of communication and synchronization takes more time than the actual work, negatively affecting performance.

17.3.1.1 Implementation Details

- **Logistics (Material required for the activity, etc.)** For the live class approximately 50 pennies per group (of 4-5 students) is required.
- **How to Conduct** Students are put into groups of 4-5 initially. In several steps, students are guided from constructing a sequential search where one student is timed in the task of finding the oldest penny to performing the task with multiple students participating, timing each experiment. Students should discover that adding more students to the task works for a while, but when taken to the extreme (each student having one penny) the task takes longer.
- **Instructional Material (Lectures, Videos, Assignments, etc.)** The adapted materials for on-line students are here.

17.3.1.2 Experience and Teaching Tips

We used this activity in week 1 of instruction for COSC 2030 (CS2) in the Fall of 2025 in approximately 40 minutes. This group had not had any PDC material presented to them in CS1 in the Spring semester of 2025, which was during the baseline year for our portion of this research effort. This unplugged activity was highly engaging to this small group of six students. Because the group was small, we never really got to the point of diminishing returns with adding enough students to the group to the point that performance decreases, but students could see that this would happen eventually.

17.3.2 Unplugged Coin Addition

17.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description. This unplugged activity simulates the addition of all elements in an array. The purpose of this activity is for students to recognize a situation in which there is a race condition because of a shared resource (the accumulator storage location in this example). This activity was designed to pair with the Plugged-In OpenMP Array Sum Activity (Section 17.4.1).

Prerequisites. It is helpful but not necessary if students have done the Unplugged Coin Search Activity. It is also helpful but not necessary that students have been exposed to array syntax to sum the contents of an array using an accumulation loop. It may additionally be motivating for students to have tried to naively parallelize this activity as shown below using OpenMP and discovered that the sum is not correctly calculated.

```
// THIS CAUSES A RACE CONDITION
#pragma omp parallel for
for (int i = 0; i < array_size; i++) {
    total += array[i]; // 'total' is shared and unprotected!
}
```

Learning Outcomes

Students will be able to:

- identify situations where parallel threads encounter a race condition due to a shared resource.
- describe why simple loop parallelization (like naive OpenMP directives) fails without proper synchronization and protection of shared resources.

17.3.2.2 Implementation Details

- **Logistics (Material required for the activity, etc.)** Enough containers (paper cups) with approximately 10 coins of mixed denominations (pennies, nickels, dimes, quarters) for each student participating. Ideally, there will be enough coins for each student to prevent a student from being able to look at the coins and immediately know how much they sum to, but not enough so that they will depart for the vending machine.
- **How to Conduct** Students should be in groups of 4-5. Each group should be given a cup with coins and 4 empty cups to use to distribute coins to team members for the parallelization. The problem is how to distribute the work of figuring out how much money is in the group. Students will recognize that dividing the coins among group members and having each student count their cup should decrease the time needed to figure out the cumulative sum. However, students should be reminded that the loop to add the contents of an array, looks like:

  ```
  total = 0
  for each value in the array
          total = total + value
  ```

  The important line to point out is that before the current element of the array can be added to *total*, the process needs to have access to the current value of *total*. To simulate this, have one student (or the instructor if the class is small) act as the keeper memory location for total, in charge of giving the total to each process (student) and recording the new value as each student accesses it. Each process needs to request the current contents of total, add the current value to it and then store it back in total. Students should observe that total is not being accumulated correctly as each student counter competes to get the current contents of total and then store their updated value.
- **Instructional material (Lectures, Videos, Assignments, etc.)** A video of the activity may be helpful for asynchronous online students, but synchronous on-line

students may be able to participate if warned ahead of time to have coins available to use. Materials for this activity are found here.

17.3.2.3 Experience and Teaching Tips

So far, the instructor has tried this activity with in-class students in a CS1 class (Spring 2026) and a CS2 class (Fall 2025). Both times, the most difficult part was getting students to understand the directions surrounding obtaining and updating the *total* accumulator variable. Both class sizes were small, so the instructor acted as the storage location for this variable. Students emailed the instructor to request the value of the accumulator, the instructor sent back the value, the student did the addition and sent back the new value to store. The concept was to create an email trace that the class could follow after the experiment. Unfortunately, students were confused and only a couple of them understood what to do. Future attempts might include changing the instructions to include values written on a paper instead of coins, or a more complicated arithmetic that would slow down the students' tasks. Future attempts might also have the total stored on a small whiteboard that could be requested from the keeper of the accumulator. Even though this is work in progress, in both classes that this activity has been tried, students ultimately seemed to understand the problem with the shared accumulator resource and were able to also realize that the simple solution of "locking" the accumulator so only one process could use it would not result in any performance gains. Students were able to conclude that each process needed to have a separate subtotal for it's section of the array and these separated subtotals would be added to a locked total. This activity took approximately 30 minutes.

17.3.3 Fork-Join Activity

17.3.3.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description This is a short activity, which was adapted from the KNOX materials (§10.4.2). We encouraged students to do this activity in pairs or small groups, students are given several precedence graphs and asked to figure out what the shortest number of time units that are required for two processors to complete the work and also the worst case number of time units, assuming no processor is voluntarily idle. A sample of one of the questions is shown in Figure 17.1. For this question, students should determine that the worst case for two processors would be 5 time units and the best case would be 4 as shown in Figure 17.2.

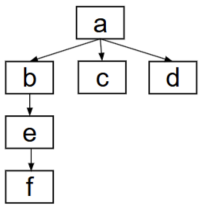
Prerequisites Awareness of the existence of multi-processor CPU architecture.

Learning Outcomes

Students will

- be introduced to the idea of separating a program into parts that can run on different cores.
- be introduced to the terminology of fork-join.

For the following dependency graphs, assume each strand (box) runs in one time unit. Assuming there are 2 processors, what is the smallest number of time units that the computation could take. What is the maximum number of time units the computation could take. Justify your answer.



A.

Figure 17.1: CASPER CS2 Example of Fork-Join Materials.

worst

Wprocesser – time units>>	1	2	3	4	5
1	a	d	b	e	f
2	-	c	-	-	-

best

Wprocesser – time units>>	1	2	3	4
1	a	b	e	f
2	-	c	d	-

Figure 17.2: CASPER CS2 Example of Fork-Join Solution for problem in Figure 17.1.

- be able to analyze a dependency graph to determine the best and worst case scenarios for a 2 processor system.

17.3.3.2 Implementation Details

- **Logistics (materials required for the activity, etc.)** The materials needed are the handout for the students.
- **How to Conduct** These materials could be used as in-class work or as a homework assignment.
- **Instructional material (Lectures, Videos, Assignments, etc.)** The materials used for this activity are available here.

17.3.3.3 Experience and Teaching Tips

The students were given time in class (about 15 minutes) to work this assignment with their team. The students seemed to be engaged in the activity and it generated a good discussion. Six students in the class submitted work for this assignment. The hardest problem turned out to be B. Only one group correctly answered that the worst case scenario for B was 7 time units. The next time we use these materials, we will try to allocate at least 30 minutes so that groups can compare answers to generate more discussion.

17.4 New Plugged-in Activities

17.4.1 Plugged-In OpenMP Array Creation Activity

Problem Description In this activity students are asked to write a C++ program that allocates space for a large array and then fills the array with the same value as the array index. The students are shown how to use the `#pragma omp parallel for` to parallelize the loop and are instructed to figure out how many different threads are created and which threads are handling which indices. The students are introduced to the use of the `chrono` library for timing code, so that the speedup can be documented. These materials were adapted from TNTECH's Plugged-In OpenMP Data Parallelism Activityin Section 2.4.2.

Prerequisites: Experience with C++ arrays and dynamic memory allocation using pointers.

Learning Outcomes

Students will:

- define and explain parallel computing terms: parallel processing, sequential processing.
- identify the number of cores on their particular machine.
- learn to use the OpenMP `#pragma omp parallel for` to do data parallelization
- learn how to compile using the `-fopenmp`.
- observe how the loop is split into chunks that are handled by different threads.
- use the C++ `chrono` library to measure and compare the execution time of sequential versus parallelized code.

17.4.1.1 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** Materials are found at here.
- **How-To Guides** In the materials created for this activity, students are guided in a step-by-step way toward learning about parallelizing a for loop using OpenMP and also guided toward use of the `chrono` library for benchmarking code. Students are encouraged to work in pairs to promote discussion as they work on this assignment.

  Part 1 of the assignment asks the students to do some preliminary research on PDC-related vocabulary and also some vocabulary specific to OpenMP to acquaint them with important terms. Students are allowed to converse with an AI about these topics and are encouraged to discuss these topics with their partner. Students also do an exploration to find out how many cores are on their own computer and on the classroom computers.

  Part 2 of the assignment students are instructed to write a sequential program to construct and initialize a large array. The goal of this task is for students to write a program that takes a noticeable amount of time to complete. Prior to this activity, students may not have written any programs that did not appear to run almost instantly.

This activity also requires students to use pointers to dynamically allocate memory, since there may not be enough memory for stack allocation of a large array. The next steps walk the students through thinking about whether or not this task could be done in parallel and how to implement this using the appropriate `#pragma` from OpenMP. In this step, the students are guided to use a smaller array so that they can document via print statements which threads are performing which array index initializations, which should ideally be the same as the number of cores that their computer has (which students should have determined in Part 1).

Part 3 of this assignment is for the students to learn to use the `chrono` library to benchmark the parallel and sequential code. They should find that there is speedup in the parallel version, but not as much as they predicted. Students are asked to reflect upon why the optimal speedup is not achieved. In this step, students are encouraged to spend some time using AI tools to learn about the `chrono` library to figure out how to use it for this application.

17.4.1.2 Experience and Teaching Tips

This exercise would work well as a 2 hour in-class lab. Another option would be for students to complete Part 1 of the activity before class so they would be prepared to start the coding part in class.

17.4.1.3 Data and Analysis

Five of the six students in this class submitted work for this assignment, but there were really 2 groups and one individual working on it. All students were able to turn in proficient work but many students did not entirely complete the assignment. For example few students gave convincing answers to why they chose the particular `chrono` clock. Also few students submitted the actual times measured for the sequential and parallel programs. In the future, this assignment will be changed to specifically ask for several measurements in a table.

17.4.2 Plugged-In OpenMP Array Sum Activity

17.4.2.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description Students are instructed to write a C++ program to add all of the elements of the array and then to parallelize that loop using OpenMP, as learned in the Plugged-In OpenMP Array Creation Activity shown in Section 17.4.1. They will make the discovery that this does not work as desired and the accumulator (which is being shared by all the threads) is a shared resource that is experiencing a race condition. Then, students participate in the Unplugged Coin Addition Activity shown in Section 17.3.2 that is a demonstration of why the naive use of the `#pragma omp parallel for` may not work as desired.

Prerequisites Students should have already worked with arrays in C++ and be able to write a loop that sums the contents of an array. The Plugged-In OpenMP Array Creation

Activity (shown in Section 17.4.1) is the prerequisite; The Unplugged Coin Addition Activity (shown in Section 17.3.2) was designed as a co-requisite.

Learning Outcomes

Students will learn:

- to recognize a race condition.
- to recognize a shared variable.
- be able to describe why a simple loop parallelization using `#pragma omp parallel for` may fail

17.4.2.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** Repository with assignment and code is here.
- **How-To Guides** This activity is designed with multiple parts including: (1) student pair programming exploration, (2) an unplugged group activity (Unplugged Coin Addition Activity, Section 17.3.2), and (3) an instructor led demonstration.

  Part 1 of the assignment asks students to write and then parallelize code using `#pragma omp parallel for` to sum the contents of an array. Since at this point, they only know how to parallelize a for loop that does not take into account shared variables (as in the Plugged-In OpenMP Array Creation Activity from Section 17.4.1), they will try it this way and discover that this does not work, leading to questions about why this isn't working as desired.

  In Part 2, students participate in the Unplugged Coin Addition Activity (see Section 17.3.2) which will illustrate that the naive approach to summing the array with a shared variable won't work in parallel. Ideally students will realize that this code can be parallelized if each of the threads are responsible for finding a partial sum of part of the array and the accumulator that stores the final result must be protected and only accessed by one thread at a time.

  Part 3 of this activity is a demonstration explaining how to implement this idea using OpenMP. The key part of the code is shown in Figure 17.3

17.4.2.3 Experience and Teaching Tips

This activity could be split up over several class sessions (completing one of the parts each class, for example), or it could be done in one class period, especially if the class is at least 90 minutes. Part 1 of the activity could be done outside of class so that class time could be used for discussion. The unplugged group activity will be difficult for on-line students to replicate, so on-line students may need to watch a video of the live activity or if possible, they could likely participate synchronously but remotely via video conference.

Figure 17.3: CASPER CS2 Part 3 of Plugged-In OpenMP Array Sum Activity

```
#pragma omp parallel //denotes the start of a parallel region
{

    long long int partialSum = 0;  //each thread will have it's own copy


    # pragma omp for  //use this inside a parallel region instead of "pragma omp parallel for"
    for (long long int j = 0; j < N; j++) {
            partialSum  += array[j]; //each thread computes a local sum
    }

    #pragma omp critical   //marks code that only one thread at a time can use
    {
            sum += partialSum;
    }
}
```

17.4.2.4 Data and Analysis

Four of the six students turned in work for this assignment. Students seemed to have gotten the main idea of the assignment. At least one student was attempting to use CodeHS as their coding environment, which does not support OpenMP. But the `std::thread::hardware_concurrency()` value printed out within the CodeHS environment is 2, so the student assumed they were using 2 cores. When the execution times came out similar between the sequential and parallel execution times, the student concluded that the overhead of parallelizing the loop was causing the lack of performance improvement. Because of the complicated behavior surrounding `pragma`, which is unfamiliar to CS2 students, this student did not do any further investigation. This was not the only student to not fully investigate the concept of the `pragma` directive. In the future, more scaffolding about `pragma` should be provided for the OpenMP assignment. Alternately, these coding assignments could be re-written using the C++ standard thread library (`std::thread`) which should work with C++11 and later without which works with any C++11-compliant compiler without additional library installations or compiler configuration, including cloud-based environments such as CodeHS. The other advantage of the (`std::thread`) libraries is that students would have more explicit control over thread execution than is possible with `OpenMP`'s high-level compiler directives.

17.4.3 Greenfoot Activity

17.4.3.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description Greenfoot is a Java-based game development platform designed specifically for education. The PDC-related objective was to expose students to Event Driven Programming. Students were given two preliminary assignments that served as a guided introduction to Greenfoot. The third part of the experience was a self-directed, novel game program that the students completed in teams of 2 or 3. The materials used were adapted from the Knox materials as described in Section 1.4.2.

Prerequisites Although Greenfoot is Java-based, these students who were in the first half of the second semester of using C++ (CS1 is also a C++ class at Casper College) were able to easily understand the Java used in Greenfoot.

Learning Outcomes

Students will:

- gain exposure to Java programming language for a non-Java CS program.
- apply fundamental OOP concepts in action including the use of inheritance in classes.
- learn to use the Greenfoot API framework.
- apply elements of Event Driven Programming.
- work collaboratively.

17.4.3.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** The instruction materials are available here.
- **How-To Guides** The Greenfoot material was used as a midterm project and was presented to students in 3 phases. The first phase was just an exploration to make sure students could get the environment to work with their own equipment and included a starter project (hedgehog). In the second phase, students were instructed to view an introductory video and then attempt to analyze the hedgehog code and make some changes to the game. The final phase of this project was a short, team project (suggested time 5 hours per teammate) to produce a game in the Greenfoot environment. The directions were intentionally open ended for this project, because it was a project new to the course and we had a desire to keep the expectations to a reasonable level.

17.4.3.3 Experience and Teaching Tips

Including this project at the CS2 level meant that the students could easily learn the Java they needed to know and all the teams were able to create games that worked reasonably well and they were clearly having a lot of fun doing this project. In weeks 6-9 (in a 16 week semester) students formed two teams of 2 and one team of 3 and were instructed to spend at least 5 hours per student either simultaneously or asynchronously creating something that would impress the rest of the class. Students were instructed to turn in time-logs and also give a presentation on their final product. Most students spent more than the required 5 hours on the project and were able to make a lot of progress in a fairly short time. The students also enjoyed showing off their projects during class presentations.

17.4.3.4 Data and Analysis

This activity was a high impact activity in this CS2 course. In addition to introducing the PDC topic of event handling, this activity reinforced CS2 topics such as class inheritance, and gave students an experience with working with a novel API and language (most students had little or no prior Java experience). In addition, this activity gave students a meaningful collaborative experience, which is a core Program Learning Outcome for the Computer Science program at Casper College. In the post course survey in answer to the question "What specific changes or adjustments should be made to enhance the overall learning experience in this course", one student left a comment that referenced this activity in a

positive light, saying "A bit more interactive stuff like the Java game creation, that was cool."

17.4.4 Plugged-In Sorting Algorithm Comparison

17.4.4.1 Problem Description, Prerequisites, and Learning Outcomes

Problem Description These materials are adapted from TNTECH's materials for CS2 (Plugged-In Sorting Algorithm Comparison, Section 11.3) as in in-class team project. The students are provided with code that implements a serial version of Quicksort, then asked to discuss with their team whether or nor Quicksort is a good candidate for parallelization. The materials gave them some hints about proceeding with the parallelization.

Prerequisites Students should have worked with `OpenMP`, `chrono`, and also have experience with the Quicksort algorithm.

Learning Outcomes

Students will:

- think critically about the Quicksort algorithm implementation in C++.
- apply the PDC concept of task parallelization.
- apply new features of OpenMP.
- practice collaborative problem solving.

17.4.4.2 Implementation Details

- **Instructional Material (Lectures, Videos, Assignments, Tests)** The materials for this activity are found here.
- **How-To Guides** Students are given a file of movie titles and starter code that reads the movie titles from movies.txt into a string vector and uses an implementation of the Quicksort algorithm in serial. In the first part of the assignment, which the live class did as an in class activity in groups of 2-3 students, the students are instructed to take a look at the code, make sure it works and that they understand it and also add in the necessary code that uses the `chrono` library to benchmark the code with the given `movies.txt` file. At the end of part 1, students are instructed to discuss with their group the question of whether the recursive calls to the `serialQuickSort` function could happen in parallel. In the second part of the activity, student are guided through the process of parallelization of the code. This was a new assignment, so we asked students to spend time on it and report progress rather than expecting them to be able to get to the answer on their own.

17.4.4.3 Experience and Teaching Tips

We normally don't do any new material on the Tuesday before Thanksgiving (the only class before a Wednesday - Friday break), but we realized that this might be the ideal time to talk about the parallelization of at least one sorting algorithm and would additionally give

us one more chance to review the PDC content we had already covered and introduce one more new piece of PDC content. Since it was the day before Thanksgiving holiday, there were not too many students (3 in person, 1 on-line). We briefly introduced the activity and then let them take a look at the QuickSort code with their partner. We then had a group discussion about whether or not this algorithm was a good candidate for parallelization. We gave them some hints on how to use OpenMP to parallelize the code and let them use AI tools to get additional help. We had group discussion at regular intervals as they were working so they could share discoveries. We kind of ended up using a "best ball" approach in that during the group discussions in which I guided the discussion toward adopting the approach taken by the group that was going in the most promising direction at each junction that we stopped to discuss progress. In the span of the 2-hour class, they were able to write a parallel QuickSort with instructor and generative AI assistance.

17.4.4.4 Data and Analysis

This activity worked very well for a live class, it might be difficult to adapt for an on-line asynchronous student.

17.5 Course-wide Pre- and Post- Course Surveys

Course-wide pre- and post-surveys were given to students in the Fall 2024 semester (No PDC interventions) and Fall 2025 (PDC interventions as described) semesters. As mentioned previously, these were not large classes. The Fall 2024 semester had 16 students. Of this population, 9 students completed the pre-course survey and 7 students completed both the pre-course survey and the post-course survey and also gave consent for data to be used for this study. The Fall 2025 semester had 6 students. In this class, 4 students completed the pre-course surveys and 2 both pre- and post-course surveys and gave consent. The results of the surveys are summarized in Tables 17.3 - 17.7.

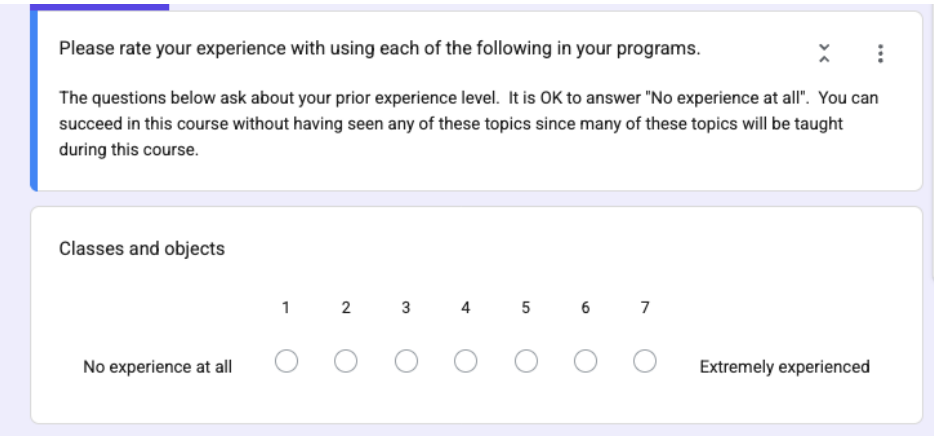


Figure 17.4: CASPER CS2 Sample Survey Question Reflecting on Level of Experience

Table 17.3 shows the survey results for some typical CS2 topics. The topics that were covered were covered in both the intervention and non intervention classes are shown high-

Table 17.3: CASPER CS2 Survey Results for Experience Questions

	No intervention			Intervention		
Experience	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdif_int$
Classes and objects	4.29	6.29	2.00	3.00	5.50	2.50
Searching algorithms	3.86	5.00	1.14	2.50	4.00	1.50
Sorting algorithms	4.29	5.29	1.00	2.50	3.50	1.00
Recursion	3.71	5.43	1.71	2.00	4.00	2.00
Template classes	2.83	5.33	2.50	2.50	4.00	1.50
Lists/Linked lists	3.43	5.71	2.29	4.00	4.50	0.50
Stacks	2.43	5.14	2.71	1.00	5.00	4.00
Queues	1.71	5.00	3.29	1.00	5.00	4.00
Priority queues	1.43	4.29	2.86	1.00	4.00	3.00
Hashing / Hash tables	1.43	4.43	3.00	1.50	3.50	2.00
Trees	1.86	5.43	3.57	1.50	4.00	2.50
Binary search trees	1.71	5.29	3.57	1.50	4.50	3.00
Balanced trees / AVL trees	1.14	4.71	3.57	1.00	3.50	2.50
Heaps	1.14	4.57	3.43	1.50	3.50	2.00
Treap	1.14	3.29	2.14	1.00	2.00	1.00
Graphs	1.57	3.43	1.86	1.50	2.00	0.50
B-Trees	1.14	2.86	1.71	1.00	3.00	2.00
Sets	2.00	5.43	3.43	1.00	4.50	3.50
Map	3.00	5.86	2.86	2.50	4.50	2.00
Writing test cases	3.57	5.71	2.14	1.50	4.00	2.50
Algorithm efficiency	1.43	4.14	2.71	2.00	4.00	2.00
Choosing data structure	3.71	4.57	0.86	2.00	4.00	2.00
Writing classes	3.86	6.29	2.43	3.00	5.00	2.00
Problem decomposition	4.71	5.57	0.86	3.50	4.50	1.00

lighted in gray. The raw responses (on a 7-point Likert scale) for the pre- survey R_{base_pre} and post survey R_{base_post} during the baseline Fall 2024 semester are shown in the first two columns for questions in which students were asked to rate their experience with various CS2 topics. The mean of the difference of the post-pre-matched survey responses are in the columns labeled $Mdif_base$ for the baseline, non-PDC-intervention class and $Mdif_int$ for PDC-intervention class. The number of responses for the baseline year with pre and post matches was 7. The last three columns show the responses and the mean difference for the Fall 2025 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre- and post matching for the intervention semester was 2.

Changes to student perception of learning in non-pdc topics. An important question that should be asked is whether inclusion of PDC topics has resulted in a decrease in perceived learning gains for the core CS2 topics. Figure 17.3 summarizes the survey results for questions that asked students about their experience using various programming languages and programming skills and concepts in their programs. A sample question is shown in Figure 17.4. For each of the topics, the students reported perceived growth overall. Topics that were not discussed (in white) were still rated higher than those specifically discussed in the class (in gray). This could be attributed to students having encountered some of these topics in another class or at least recognizing most of the terms in the survey at the end of the course, so being more optimistic about their overall level of experience. The intervention group showed gains, but some of the areas did not show the same magnitude of gains as the non-intervention group. For example, the "Lists/Linked lists" category which

did not show significant growth, although this is a topic heavily covered in this class. In looking at the individual responses, one student rated their level of experience in both the pre- and post- survey as a "6", the other student rated their level of experience a "2" in the pre-survey and "3" in the post survey. For this topic it is clear that one student's response skewed the overall average, which is to be expected for this small sample size. The other block of material that has noticeably lower $Mdif$ values is the Hashing - Heaps section of the survey. These topics were covered in weeks 14-15 of the course in which the only new PDC material added was the Plugged-In Sorting Algorithm Comparison-based in-class activity. It is possible that this diversion did add to the cognitive load of the course in such a way as to diminish perceived learning for these traditional CS2 topics. One student gave the following feedback to the question "What specific changes or adjustments should be made to enhance the overall learning experience in the course:"

> There were quite a few things we worked on for only a week. It would've been cool if those things came back up so there was more repetition in our learning. When we're learning so much new stuff, it feels like one week with a new concept is not enough to set it in stone.

In contrast, the comments from students in the non-intervention semester were generally positive about the course. Given the small number cohort with only 2 matched responses these results are reported as a descriptive exploratory observation with the understanding that individual student perceptions and external workload factors heavily influence average Likert scores in this data.

The summary of survey results that asked students about experience with PDC topics is shown in Table 17.4. In this table, raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post survey R_{base_post} during the baseline Fall 2024 semester are shown in the first two columns for questions in which students were asked to rate their experience with programming languages and the use of various programming topics in their programs. The mean of the difference of the post-pre matched survey responses are in the columns labeled $Mdif_base$ for the baseline, non-PDC-intervention class and $Mdif_int$ for PDC-intervention class. The number of responses for the baseline year with pre and post matches was 7. The last three columns show the responses and the mean difference for the Fall 2025 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 2. Encouragingly, the PDC intervention class showed increases in perceived gains on this topic that were similar to the gains reported for the standard CS2 topics.

Table 17.4: CASPER CS2 Survey Results Experience with PDC Topics

	No intervention			Intervention		
Experience	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdif_int$
Parallel processing	2.57	3.00	0.43	2.00	3.50	1.50
Distributed computing	2.00	2.57	0.57	1.50	3.00	1.50
Event handling	2.71	3.57	0.86	2.00	4.50	2.50

Students were also asked to rate their level of interest in computing and in PDC. These survey questions are shown in Figure 17.5 and a summary of the results are shown in Table 17.5.

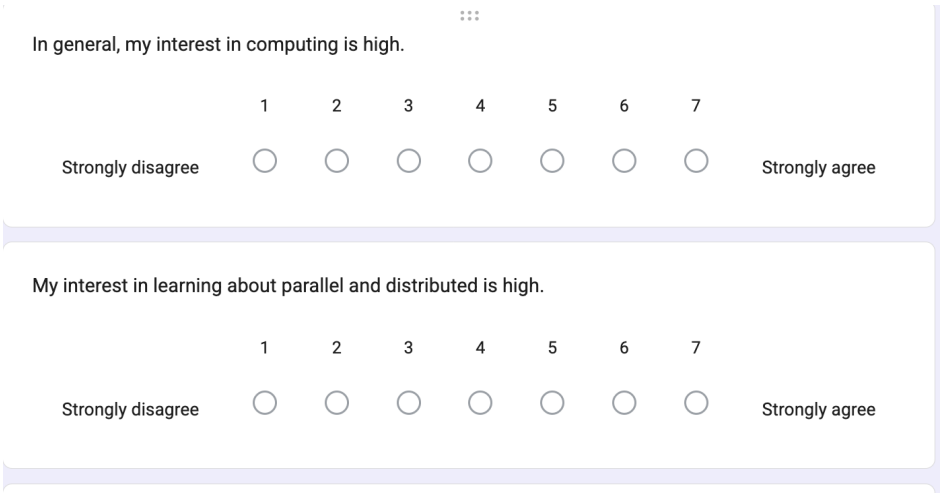


Figure 17.5: CASPER CS2 Sample Survey Question Reflecting on Level of Interest in Computing and PDC

Table 17.5: Survey Results for CS2 Interest Questions- Casper College

	No intervention			Intervention		
Interest	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdif_int$
Computing	6.29	6.29	0.00	7.00	6.00	-1.00
PDC	6.00	6.00	0.00	4.00	3.50	-0.50

For Table 17.5, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post survey R_{base_post} during the baseline Fall 2024 semester are shown in the first two columns for questions in which students were asked to rate their Interest in computing and interest in PDC topics. The mean of the difference of the post-pre matched survey responses are in the columns labeled $Mdif_base$ for the baseline, non-PDC-intervention class and $Mdif_int$ for pdc-intervention class. The number of responses for the baseline year with pre and post matches was 7. The last three columns show the responses and the mean difference for the Fall 2025 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 2. These results are discouraging since they appear to show a decrease in interest in both computing and PDC topics for the PDC intervention class and no change in the baseline year.

Table 17.6: CASPER CS2 Survey Results Interest Questions

	No intervention			Intervention		
Importance	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdif_int$
Importance - PDC	5.57	6.29	0.71	5.50	4.50	-1.00

For Table 17.6, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Fall 2024 semester are shown in the first two columns for questions in which students were asked the level of agreement with the following statement: "In general, I believe learning about parallel and distributed computing is important." The mean of the difference (post-pre) of the matched survey responses are in

the columns labeled $Mdif_base$ for the baseline, non-PDC-intervention class and $Mdif_int$ for PDC-intervention class. The number of responses for the baseline year with pre and post matches was 6. The last three columns show the responses and the mean difference for the Fall 2025 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 2. Again the overall responses for the PDC intervention class are disappointing in that each of the two students dropped their enthusiasm by one point between the pre-survey and the post survey. Although, we should again note the small sample size for the PDC-intervention group with pre- and post-matched surveys.

Table 17.7: CASPER CS2 Survey Results for Understanding Questions

	No intervention			Intervention		
Understanding	R_{base_pre}	R_{base_post}	$Mdif_base$	R_{int_pre}	R_{int_post}	$Mdif_int$
Computing	4.71	5.57	0.86	3.50	4.50	1.00
PDC	3.29	3.86	0.57	1.50	4.50	3.00

For Table 17.7, the raw responses (on a 7-point Likert scale) for the pre-survey R_{base_pre} and post-survey R_{base_post} during the baseline Fall 2024 semester are shown in the first two columns for questions in which students were asked their level of agreement with the following statements: "I believe I have a good understanding of computing" and "I believe I have a good understanding of parallel and distributed computing." The mean of the difference (post-pre) of the matched survey responses are in the columns labeled $Mdif_base$ for the baseline, non-PDC-intervention class and $Mdif_int$ for PDC-intervention class. The number of responses for the baseline year with pre and post matches was 7. The last three columns show the responses and the mean difference for the Fall 2025 semester in which the PDC interventions described in this chapter were implemented. The number of responses with pre and post matching for the intervention semester was 2. Students in the PDC intervention group here reported large growths in their perceived understanding compared to the no intervention group.

17.6 Conclusion

Lessons learned The experience of implementing PDC concepts in this chapter highlights a distinct trade-off between conceptual engagement and technical implementation hurdles. The unplugged activities, such as the Unplugged Penny Activity (Section 17.3.1) and Unplugged Coin Addition Activity (Section 17.3.2) were highly engaging to students. These exercises were also not difficult to prepare or implement and did not require a lot of class time. They were successful at building an intuitive foundation for fundamental PDC concepts and vocabulary. These activities generally rate lower in the Bloom's taxonomy such as "Remember" and "Understand". In our experience, a major difficulty in presenting PDC material at the CS1/CS2 level is transitioning to higher impact assignments that reach higher Bloom's levels such as "Analyze" or "Evaluate". The OpenMP-based activities were used to attempt to move to higher Bloom's levels. The OpenMP based activities had a steep learning curve that necessitated and required much more instructional support.

Because of the logistics of this project, the Casper College test site implemented PDC intervention to cohorts of both CS1 students and CS2 students who had no prior PDC interventions in our CS courses. We were able to observe many of the same PDC interventions in both courses including Unplugged Penny Activity, Unplugged Coin Addition, Plugged-In OpenMP Array Creation, Plugged-In OpenMP Array Sum. Overall, the interventions were significantly easier to introduce to a CS2 student and students seemed to be more academically successful than the CS1 population. From an instructor's perspective, many of these activities that were originally designed for CS1 seemed to be working well for the CS2 audience; students appeared to be getting the concepts more readily and deeply than the CS1 audience. One potential hypothesis for the lower post-survey interest scores in this small intervention cohort is the heightened cognitive load resulting from frequent new weekly PDC topics alongside standard CS2 material.

Future Work

For our particular context at Casper College we need to carefully consider how to make room for PDC topics in this CS2 course. Deeper consideration needs to be taken in figuring out if there is more material that can be safely eliminated from the core of CS2 to make more room for the PDC material. As mentioned previously, in the experience reported here, many of the same interventions used in CS1 were also used here. Going forward, we might consider doing fewer interventions in both CS1 and CS2 or splitting up the interventions and doing primarily unplugged activities in CS1. One assumption that we made in designing the interventions was that combining an unplugged activity with a plugged-in activity would produce maximum impact for students in terms of student learning outcomes. This assumption should be further examined, given that several test sites in this study had good results with purely unplugged/simulation activities. Alternatively, we could do Unplugged Penny Activity and Plugged-In OpenMP Array Creation in CS1 and have Unplugged Coin Addition and Plugged-In OpenMP Array Creation in CS2. The concern we see with this approach is the overhead in reviewing both OpenMP and the `chrono` library needed for benchmarking. We also plan to explore the use of `std::thread` instead of `OpenMP` for the C++ Plugged-in assignments to mitigate both the environmental issues and the lack of explicit control over threads with `OpenMP`.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter17-Casper

Detailed Pre and Post Syllabus

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter17-Casper/Detailed_Pre_and_Post_Syllabus

Organization of Material

Materials in the Github repository are organized as follows;

- Detailed_Pre_and_Post_Syllabus – This directory contains information on the baseline semester of the course before the PDC interventions were added (Fall 2024) and the post semester when the PDC interventions were added (Fall 2025).
- Evaluation_Data_and_Analysis – This directory has documentation concerning the instruments and data collected during the course of this effort.
- Plugged_Activities – This directory contains resources used to implement the PDC Activities that involved coding activities.
- Unplugged_Activities – This directory contains resources used to implement the PDC Activities that did not involve coding.

Survey and Other Anonymized Data and Analysis

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter17-Casper/Evaluation_Data_and_Analysis

Chapter 18

CS2 Course Package – Montclair State University

***Java Infusion through Event-Driven Greenfoot Game and Parallel Image Processing*†**

Michelle Zhu[1,2] and Jiayin Wang[1]

[1]Montclair State University [2]Kennesaw State University

†**This chapter appears in the CDER Center e-book:** Prasad, S. K., Weems, C., Thota, N., Sussman, A., Vaidyanathan, R., Gannod, G., Crockett, A., Bunde, D., Spacco, J., Srivastava, S., Bourke, C., Suo, X., Maher, P., Gruner, C., Smith, M., Zhu, M., and Wang, J. Nov. 2026. *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* First Edition. CDER Center. NSF Award #2321015-20. https://doi.org/10.14809/20.500.12588/7992.2.

© 2026 CDER Center and the chapter authors. Unless otherwise noted, this work is made available for educational use with attribution.

Chapter contents

18.1 Introduction . . . 487
18.2 How CS2 was changed and PDC topics integrated . . . 487
18.2.1 Integrated Syllabi (Pre and Post) . . . 488
18.2.2 Highlights . . . 489
18.2.3 Challenges . . . 489
18.3 New Plugged Activities . . . 490
18.3.1 Event-Driven Java Programming Through Greenfoot Game Design . . . 490
18.3.1.1 Problem Description, Prerequisites, and Learning Outcomes . . . 490
18.3.1.2 Implementation Details . . . 491
18.3.1.3 Experience and Teaching Tips . . . 491
18.3.1.4 Data and Analysis . . . 492
18.3.2 Multithreaded Java Computing Through Parallel Image Processing . . . 493
18.3.2.1 Problem Description, Prerequisites, and Learning Outcomes . . . 493
18.3.2.2 Implementation Details . . . 493
18.3.2.3 Experience and Teaching Tips . . . 494
18.3.2.4 Data and Analysis . . . 495
18.4 Course-wide Pre and Post Course Surveys . . . 498
18.5 Other Activities and Ideas . . . 505
18.6 Conclusion . . . 505
Appendix . . . 506

Chapter figures

18.1 A Student Sample of a Greenfoot Game Design . . . 492

Chapter tables

18.1 Relevant PDC topics and Bloom's Levels . . . 484
18.2 MSU CS2 Gender Distribution . . . 485
18.3 MSU CS2 Race/ethnicity Distribution . . . 485
18.4 MSU CS2 Current Class Standing Distribution . . . 485
18.5 Summary comparison of CS2 before and after PDC integration. . . . 488
18.6 Post-integration CS2 course schedule with adopted PDC modules highlighted. . . . 489
18.7 Coding of seven-point survey response scales . . . 495
18.8 Pre- and post-activity understanding of PDC concepts in the Parallel Image Processing activity. . . . 496
18.9 Student perceptions of the Parallel Image Processing activity. . . . 497
18.10 Themes from open-ended Parallel Image Processing responses. . . . 498
18.11 CS2 survey results for traditional programming concepts. . . . 500
18.12 CS2 survey results for PDC-related concepts. . . . 501
18.13 CS2 survey results for attitudes and beliefs. . . . 502
18.14 Matched pre- and post-course survey results for traditional programming concepts in CS2. . . . 503
18.15 Matched pre- and post-course survey results for PDC-related concepts in CS2. . . . 503
18.16 Matched pre- and post-course survey results for attitudes and perceptions in CS2. . . . 504

PDC Course Material Used in CS2 at Montclair State University

Abstract

This chapter presents two plug-in learning activities that integrate parallel and distributed computing (PDC) concepts into CSIT 112, Fundamentals of Programming II, a CS2-level Java programming course at Montclair State University. The first activity introduces students to event-driven programming through a Greenfoot game development lab, where they implement mouse and keyboard interaction within a simulated game environment. The second activity applies data-parallelism through Java multithreading to image processing kernels, allowing students to implement and observe the effects of edge detection, blurring, and salt-and-pepper noise removal with parallel threads.

Together, these activities expose students to foundational PDC concepts, including speedup, Amdahl's Law, thread synchronization, and concurrent execution, while reinforcing core event-handling principles through interactive game design. By pairing a visually engaging game development exercise with a computationally intensive image processing task, the module motivates students to connect abstract PDC theory with tangible, observable outcomes: faster image transformations through multithreading and responsive, real-time game interactions through event handling.

Each activity is supported by a pre-activity survey, a post-survey quiz, and hands-on lab to assess learning outcomes and student perceptions of PDC concepts. Results from paired pre/post survey responses indicate measurable gains in students' self-reported understanding of parallel processing, distributed computing, event-driven programming, and overall PDC-related confidence. These findings suggest that even brief, well-scaffolded plugged interventions, grounded in familiar tools like Java and Greenfoot, can provide meaningful early exposure to PDC concepts within a CS2 curriculum, helping bridge the gap between introductory programming and more advanced parallel and distributed systems coursework.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Revel for Java Software Solutions, Foundations of Program Design, 10th Edition by Lewis & Loftus[37]

Relevant core courses: CS2: CSIT 112, Fundamentals of Programming II

Pre-requisites: CS1: CSIT 111, Fundamentals of Programming I

Relevant PDC topics: Table 18.1 lists the integrated PDC topics with bloom's classification (Remember (RE), Understand (UN), Apply (AP), Analyze (AN), Evaluate (EV), and Create (CR)).

Table 18.1: Relevant PDC topics and Bloom's Levels

PDC Concept	Chapter Section	
	18.3.2	**18.3.1**
Speedup	RE, UN, AP, AN, EV	
Thread Synchronization	RE, UN, AP, AN, EV, CR	
Event Handling		RE, UN, AP, CR

Learning outcomes:

1. **Object-Oriented Programming (OOP):** Deepen knowledge of Java classes, objects, encapsulation, inheritance, and polymorphism.
2. **Algorithm Development & Advanced Logic:** Create robust algorithms using recursion, subroutines, and structured user functions.
3. **Data Management & I/O:** Write programs capable of file manipulation, text processing, and storing data using complex arrays.
4. **Problem Solving:** Break down complex computational problems and develop step-by-step solutions with proper testing and documentation
5. **Parallel and Distributed Computing**: Articulate core parallel computing concepts including speedup, Amdahl's Law, and thread synchronization, and apply them by implementing and empirically evaluating multithreaded Java programs and designing event-driven Java programs that respond to asynchronous mouse and keyboard input in an interactive environment.

Context for use: Montclair State University, located 12 miles west of midtown Manhattan, is New Jersey's largest Hispanic-Serving Institution (HSI) and a public R2 doctoral research university enrolling approximately 23,000 students. The learning modules were implemented in CSIT 112, *Fundamentals of Programming II*. CSIT 112 is a CS2-level introductory Java programming course offered by the School of Computing and required for both the BS in Computer Science and the BS in Information Technology. It serves as the second course in the introductory programming sequence. Students enter the course after CSIT 111, *Fundamentals of Programming I*, with prior exposure to variables, expressions, classes, conditionals, and loops. The course is taught over a 14-week semester, with approximately five sections offered each semester and an average class size of 32 students. Table 18.2, Table 18.3, and Table 18.4 summarize the demographic characteristics of these respondents by gender, race/ethnicity, and current class standing, respectively. These tables provide context for comparing student backgrounds across the two CS2 offerings.

Table 18.2: MSU CS2 Gender Distribution

Gender	Fall 2024 ($n = 15$)	Fall 2025 ($n = 46$)
Female	7	14
Male	8	30
Other	0	0
Not reported	0	2
Total	15	46

Table 18.3: MSU CS2 Race/ethnicity Distribution

Race/Ethnicity	Fall 2024 ($n = 15$)	Fall 2025 ($n = 46$)
Asian / Pacific Islander	2	6
Black/African American	4	9
Hispanic	2	10
Middle Eastern or North African	0	4
White	6	11
Declined to specify	1	4
Not reported	0	2
Total	15	46

Table 18.4: MSU CS2 Current Class Standing Distribution

Current Class Standing	Fall 2024 ($n = 15$)	Fall 2025 ($n = 46$)
First year (freshman)	0	0
Sophomore	5	25
Junior	6	12
Senior	4	8
Not reported	0	1
Total	15	46

The pre-integration version of CS2 focused on object-oriented programming with Java, including polymorphism, interfaces, graphical user interfaces, and event handling. The post-integration version retained these topics while making PDC more explicit through outcomes on concurrent applications, data parallelism, Java multithreading APIs, speedup, and efficiency.

Two PDC-related modules were adopted. The Image Processing module introduced data parallelism through lecture material on parallel computing and Java multithreading, sample code using edge detection, and a Lab 5 box blur filter assignment in which students compared sequential and parallel implementations. The Greenfoot module was implemented as a course project, where students designed an interactive Java game using event handling. Together, these modules introduced both data parallelism and event-driven programming within the existing CS2 structure.

18.1 Introduction

As multicore and many-core architectures have become the standard in consumer and enterprise computing alike, the ability to reason about parallelism is no longer a specialized skill reserved for advanced coursework, it is a foundational competency for any modern programmer. Yet introductory computer science curricula have historically lagged behind this hardware reality, often deferring parallel and distributed computing (PDC) concepts until upper-level systems or architecture courses, long after students have internalized sequential programming as the default mode of thinking. This chapter addresses that gap by describing the integration of PDC concepts into CSIT 112, *Fundamentals of Programming II*, a CS2-level Java course at Montclair State University that serves as a direct continuation of CSIT 111, *Fundamentals of Programming I*.

The remainder of this chapter details the design, implementation, and assessment of these two activities, beginning with the Greenfoot-based event-driven game development lab, followed by the parallel image processing lab grounded in Java multithreading.

18.2 How CS2 was changed and PDC topics integrated

Prior to this work, CS2 contained very simple multi-thread Java programming without any theoretical content on parallel programming. Students progressed from CS1 to CS2 learning object-oriented design, data structures, and sequential algorithmic thinking, with no structured opportunity to engage with concurrency, parallelism, or distributed systems concepts until much later in the curriculum such as system programming and operating systems. This omission stood in increasing tension with the reality that the Java programs students were learning to write would ultimately run on multicore hardware, where understanding concepts such as thread synchronization, speedup, and parallel decomposition is essential.

Our students were not entirely unfamiliar with PDC ideas. In CS1, they had participated in an unplugged activity, a "flag maker" exercise, designed to build early, low-stakes intuition about parallel task decomposition without requiring any programming. While this activity succeeded in introducing students to the conceptual vocabulary of PDC and in convincing them of its relevance, it did not give them any hands-on experience translating these ideas into code. Students left CS1 understanding *that* parallelism mattered, but not *how* to design and implement it as programmers. Chapter 9 introduces conceptual data parallelism through the unplugged Flag Maker activity in CS1, while Chapter 18 extends this foundation to event-driven programming and multithreaded implementation in CS2. This left an open pedagogical question: how do we move students from conceptual awareness to practical fluency, using applications that are both tangible and intrinsically motivating?

This question is the impetus for the work described in this chapter. We designed and integrated two plugged activities into CS2, namely a Greenfoot-based game development lab emphasizing event-driven programming, and a Java multithreading lab applying data-parallelism to image processing kernels. Both were selected specifically for their capacity to make abstract PDC concepts visible and rewarding. Game development offers immediate, interactive feedback through mouse and keyboard input, while parallel image processing

offers a direct, visual payoff: students can literally watch an image transform faster as threads are added, making the otherwise abstract notion concrete.

18.2.1 Integrated Syllabi (Pre and Post)

The pre- and post-integration syllabi for CS2 show that the course remained a CS2-level Java programming course centered on object-oriented programming. Both versions include standard CS2 topics such as inheritance, polymorphism, interfaces, exception handling, recursion, collections, graphical user interfaces, and event handling. The main change was not a replacement of the CS2 curriculum, but the addition of explicit PDC topics and concrete PDC-related learning activities.

In the pre-integration syllabus, CS2 did not include data parallelism, multithreading, image processing, or Greenfoot. In the post-integration syllabus, PDC was added through two adopted modules. The first module introduced data parallelism through lecture material on parallel computing and Java multithreading, sample edge-detection code, and Lab 2, an image-processing assignment using a box blur filter. The second module extended the event-handling portion of the course through a Greenfoot game project, in which students designed an interactive Java game and explained how event handling was implemented.

Table 18.5: Summary comparison of CS2 before and after PDC integration.

Syllabus Element	Pre-Integration Version	Post-Integration Version
Primary language	Java	Java
Core CS2 content	OOP, inheritance, polymorphism, interfaces, exception handling, recursion, collections, GUI, and event handling	OOP, inheritance, polymorphism, interfaces, exception handling, recursion, collections, GUI, event handling, multithreading, and data parallelism
PDC module	Not explicitly included	Explicit outcomes on concurrent applications, data parallelism, Java multithreading APIs, speedup, and efficiency
Image-processing activity	Not included	**Lab 2: Data-parallel image processing with edge-detection sample code and box blur filter**
Greenfoot activity	Not included	**Greenfoot event-handling game project**
Assessment structure	Labs, REVEL exercises, in-class exercises, midterm exam, final exam, and attendance	Labs, REVEL exercises, midterm exam, final exam, attendance, in-class exercises, and **Greenfoot project**

Table 18.6 summarizes the post-integration course schedule. The highlighted rows show the two adopted learning modules: data parallelism through parallel computing and image processing, and event handling through the Greenfoot project.

Overall, the post-integration syllabus preserved the existing CS2 sequence while adding two concrete PDC-related experiences. The image-processing module connected data parallelism and multithreaded execution to a visual computing problem through lecture, edge-detection sample code, and Lab 2. The Greenfoot project extended the event-handling portion of the course into an interactive game-design experience.

Table 18.6: Post-integration CS2 course schedule with adopted PDC modules highlighted.

Module	Week	Activities	Assignments
Module 1: Introduction	Week 1	Slides: Class 1	None
Module 2: Review	Week 2	Slides: Class 2	REVEL Exercises; Lab 1
Module 3: Data Parallelism	**Weeks 2–3**	**Slides: Class 3; data parallelism; Java multithreading; image-processing examples using edge detection**	**Lab 2: PDC image processing with box blur filter**
Module 4: Polymorphism	Weeks 3–4	Slides: Class 4–6	REVEL Exercises; Lab 3
Module 5: Exception	Weeks 5–6	Slides: Class 7–8	REVEL Exercises; Lab 4
Module 6: Recursion	Weeks 6–7	Slides: Class 9	REVEL Exercises; Lab 5
Module 7: Midterm	Weeks 7–8	Slides: Class 10; Midterm Review; Midterm Exam	None
Module 8: Collection	Weeks 8–10	Slides: Class 11–13	REVEL Exercises; Lab 6
Module 9: JavaFX and Event Handling	**Weeks 10–12**	**Slides: Class 14–17; event handling; Greenfoot programming**	**Greenfoot event-handling project**
Module 10: Final Review and Exam	Weeks 13–14	Slides: Class 21–22; Final Review; Final Exam	None

18.2.2 Highlights

The CS2 integration preserved the core structure of the course. The course remained a CS2-level Java programming course focused on object-oriented programming, including inheritance, polymorphism, interfaces, exception handling, recursion, collections, graphical user interfaces, and event handling. The programming language, main course sequence, textbook platform, labs, exams, and final project structure were largely retained.

The PDC integration was achieved by reallocating existing class time rather than adding new weeks to the course. In the previous schedule, four class meetings were used for final review exercises; this was reduced to two. The JavaFX unit was also shortened from five class meetings to three by removing two classes that previously focused on JavaFX sample code for event handling. These changes freed four class meetings.

The four reallocated class meetings were used for the new PDC-related activities: one class for a lecture on data parallelism, one class for the image-processing lab, one class for an introduction to Greenfoot, and one class for Greenfoot project presentations. In this way, the course added applied experiences in data parallelism and event handling while preserving the main CS2 programming content.

18.2.3 Challenges

The integration of these activities was not without challenges. Foremost among them was the constraint of instructional time: CS2 already carries a full CS2 syllabus, and incorporating two additional labs required careful negotiation of contact hours without displacing core course content. Secondly, more procedural challenge arose from our research design. Because we sought to evaluate the activities' learning impact through paired pre- and post-activity assessments, we were required to obtain Institutional Review Board (IRB) consent from participating students. This requirement introduced an unexpected barrier to data collection: many students, despite completing the activities themselves as part of the regular course,

did not return signed consent forms, largely because participation in the *study* was perceived as optional and was not reinforced. Notably, course policy did not permit us to offer bonus points or other academic incentives for consent, and as a result, only approximately one-third of the class ultimately provided consent for inclusion in our assessment data.

These constraints shaped both the scope of our findings and our plans for future iterations of this work. Looking ahead, we intend to explore alternative strategies for improving consent participation that remain consistent with IRB and institutional policy, and we are considering proposals to formalize both activities as mandatory components of CS2 across all sections, independent of their use in any specific research study. Doing so would not only strengthen the evidentiary basis for evaluating their pedagogical impact but would also ensure equitable PDC exposure for all students in the course, regardless of their willingness or availability to participate in IRB-governed research.

18.3 New Plugged Activities

18.3.1 Event-Driven Java Programming Through Greenfoot Game Design

18.3.1.1 Problem Description, Prerequisites, and Learning Outcomes

In CS2, the Greenfoot activity was adopted as a plugged-in, open-ended programming project for teaching event handling and event-driven programming. Students first studied a sample Greenfoot game and then designed and implemented their own games individually or in groups of up to three. Each game was required to include a clear objective and interactive behaviors controlled through events such as keyboard input, mouse actions, or collisions. The completed game served as the final course project.

The activity required basic CS2-level knowledge of Java, including classes and objects, methods, inheritance, conditionals, loops, and basic graphical user interface concepts. Students did not need previous experience with Greenfoot or game development. The sample game and lecture materials introduced the Greenfoot environment, the relationship between the `World` and `Actor` classes, and the methods needed to implement movement, collision detection, scoring, lives, and win and loss conditions.

Based on the classroom implementation, the main learning goal was to help students apply event handling within a larger, creative Java program. Students were expected to explain the relationship among event sources, events, and event handlers; use the `act()` callback to define actor behavior; respond to keyboard, mouse, collision, and other game events; and manage changes in game state. The activity also helped students reinforce object-oriented programming, problem decomposition, code documentation, debugging, teamwork, and oral presentation skills. More broadly, students learned that event-driven programs are controlled by a framework that invokes their callback methods, rather than by a programmer-defined `main()` method.

For more information about Greenfoot and the original activity, please refer to Chapter 1, Section 1.4.2.

18.3.1.2 Implementation Details

The Greenfoot activity was implemented through a lecture, sample-code demonstration, and open-ended final project. The instructor first reviewed graphical user interfaces and the relationship among controls, events, and event handlers. The lecture introduced Greenfoot as a Java game engine and explained how the Greenfoot framework repeatedly invokes each actor's `act()` method. This allowed students to compare event-driven execution with the conventional execution model based on a programmer-defined `main()` method.

The instructor then demonstrated the Hedgehog sample game. In this game, the player uses keyboard input to move a hedgehog, collect food, and avoid balloons. The sample code illustrated how to use `Greenfoot.isKeyDown()` to respond to keyboard events, `setLocation()` to move an actor, and collision-detection methods such as `isTouching()` and `removeTouching()` to manage interactions among actors. Additional methods demonstrated how to update points and lives, display status information, and determine win and loss conditions. The example also showed how actor state and behavior are organized within separate Java classes.

After studying the sample code, students designed and implemented their own Greenfoot games. The project could be completed individually or in a group of up to three students. Each game was required to have a clear objective and include at least one form of event handling, such as keyboard input for movement or mouse input for game actions. Students were also required to document the event-handling mechanisms clearly through comments in their code.

The project was intentionally open-ended so that students could apply the sample code while developing an original game. Students determined the game's objective, actors, interactions, rules, visual design, and win or loss conditions. This required them to decompose the game into multiple classes and behaviors and coordinate changes in the game state in response to events. Students working in groups were required to identify each member's role and contribution.

At the end of the project, each student or group delivered a five-minute in-class presentation. The presentation included a live demonstration of the game and an explanation of how event handling was implemented and how it contributed to the gameplay. Students were also expected to answer questions from the instructor or class.

Students submitted a compressed project folder containing the complete source code, a README file, and all images, sounds, or other assets required to run the game. The README included a brief description of the game, instructions for playing it, known bugs or limitations, and the roles of individual group members. The project was evaluated based on creativity and design, implementation of event handling, functionality, code documentation, and the final presentation.

18.3.1.3 Experience and Teaching Tips

The Greenfoot project worked well as a final project because it allowed students to apply multiple CS2 concepts in a single creative programming task. Although event handling was the main PDC-related focus, students also practiced object-oriented design, inheritance, class interaction, method design, conditionals, loops, arrays or collections, debugging,

documentation, teamwork, and presentation skills. The game format gave students higher motivation than many traditional assignments; some students worked overnight, and some hand-designed the objects and visual elements used in their games.

A useful teaching strategy is to begin with a simplified sample game as a baseline. The sample should demonstrate the basic Greenfoot structure, including actors, state, the `act()` callback method, and at least one event-handling mechanism. Students can then extend the baseline with their own creative design, additional actors, rules, scoring systems, levels, or interaction patterns.

Showing examples of previous student projects can also inspire students and help them understand the expected scope. In this implementation, students created a variety of games, including a Snake-style game, a Morse-code decoding game, and a Contra-style action game 18.1. These examples showed that students were not only applying event handling, but also integrating broader CS2 knowledge into complete interactive programs.

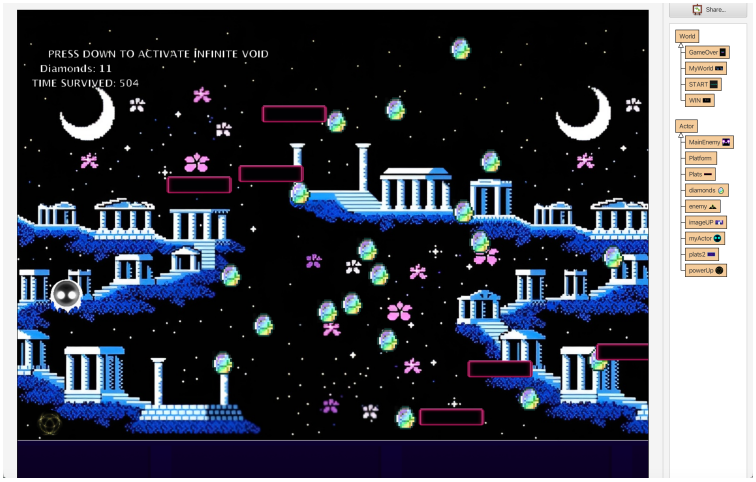


Figure 18.1: A Student Sample of a Greenfoot Game Design

The project presentation can be made more engaging by asking students to vote for their favorite games or projects and giving small awards. This encourages students to polish their games and gives the presentation session a stronger sense of audience and celebration.

Clear project requirements are important. Each game should include a clear objective, at least one event-handling mechanism, commented code, a README file, and a short presentation. In the README file, students should explicitly explain how event handling is implemented in the game, including what event is detected, where it is handled in the code, and how the game state changes in response. Overall, the Greenfoot project provides a strong culminating experience because it combines event-driven programming with the broader object-oriented programming skills developed throughout CS2.

18.3.1.4 Data and Analysis

The Greenfoot project was evaluated as a 50-point final project. The project rubric included creativity and design, event handling, functionality, code documentation, and presentation. Students were required to submit source code, include a README file, and

deliver a short presentation with a live demo and an explanation of how event handling was implemented in the game.

The grading results show strong student performance on the project. Among 45 students, 43 students earned full credit. One student did not submit the project and received a score of zero. One student completed the project but lost 15 points because the presentation requirement was not completed.

These results suggest that most students were able to complete the Greenfoot project successfully and meet the major functional and event-handling expectations. The high full-credit rate also matches the instructional experience that students were strongly motivated by the game-design format. Compared with more traditional programming assignments, the Greenfoot project encouraged students to invest more effort, creativity, and time into their work.

18.3.2 Multithreaded Java Computing Through Parallel Image Processing

18.3.2.1 Problem Description, Prerequisites, and Learning Outcomes

In CS2, the Parallel Image Processing activity was adopted as a plugged-in Java programming exercise. Students applied image-processing algorithms using sequential and multithreaded implementations and compared their correctness and performance under different thread counts and task-partitioning strategies.

The activity required basic CS2-level understanding of Java, including loops, arrays, methods, classes, objects, file input and output, and exception handling. Students did not need prior experience with parallel programming. The required multithreading concepts and Java tools, including threads, `ExecutorService`, and `CountDownLatch`, were introduced as part of the activity.

Based on the classroom implementation, the main learning goal was to develop both conceptual understanding and practical programming skills. Students were expected to distinguish sequential execution, concurrency, and parallelism; partition image data among multiple threads; identify race conditions; apply synchronization when necessary; and compare execution time and speedup across different numbers of threads. The activity also helped students connect familiar CS2 concepts to PDC concepts such as data parallelism, synchronization, task granularity, overhead, and scalability.

For more information, please refer to Section 10.4.4 for the Parallel Image Processing activity.

18.3.2.2 Implementation Details

The Parallel Image Processing activity was implemented using a lecture-demonstration-lab sequence. The instructor first used lecture slides to introduce sequential execution, concurrency, parallelism, threads, processor cores, race conditions, synchronization, speedup, parallel overhead, task granularity, and load balancing.

The instructor then demonstrated image processing using edge detection and histogram generation. The edge-detection example showed how image pixels could be divided among

multiple threads for data-parallel execution. In the histogram example, students compared unsynchronized and synchronized implementations. The unsynchronized version demonstrated how concurrent updates to shared histogram bins could cause race conditions and incorrect results, while the synchronized version showed how locking could protect the shared updates. These examples helped students understand both the performance benefits of parallel processing and the need for synchronization when threads modify shared data.

Browser-based demonstrations of grayscale conversion, edge detection, and image blur were also used to visualize pixel-level parallelism. Students observed how the image-processing work was divided among different numbers of threads and compared the resulting execution times and speedup.

The programming lab focused on a 3×3 box-blur algorithm. Students first examined and ran the sequential implementation to understand pixel traversal, neighborhood averaging, border handling, image input and output, and the timing procedure. After confirming that the sequential program produced the expected output, they implemented a parallel version that partitioned the image by rows or groups of rows. They compared the output images to verify correctness and discussed why increasing the number of threads did not always produce proportional performance improvement. The concluding discussion connected their observations to scheduling overhead, available processor cores, scalability, and the trade-off between performance and code complexity.

18.3.2.3 Experience and Teaching Tips

- Reserve two class sessions for the activity. Use the first session to introduce the PDC concepts through the lecture slides and sample programs, including the edge-detection and histogram demonstrations. Use the second session for the box-blur programming lab, performance experiments, and discussion of correctness, speedup, synchronization, overhead, and scalability. This schedule gives students sufficient time to understand the concepts before applying them in the lab.
- Use the histogram example to demonstrate race conditions and synchronization. Explain that a statement such as `hist[v]++` is not necessarily atomic because it involves separate read, modification, and write operations. Comparing the unsynchronized and synchronized results makes the need for protecting shared updates more concrete.
- Run the sequential image-processing program before introducing the parallel version. Students should first understand the box-blur algorithm, image paths, border handling, and expected output before debugging multithreading code.
- Process a batch of images rather than a single image when comparing the execution times of the sequential and parallel implementations. Processing only one image may finish too quickly for students to observe clear and consistent performance differences across thread counts. In this activity, a batch of 15 images, each approximately 11 MB, was used to create a sufficiently large workload. This made the differences among sequential execution and parallel execution with different numbers of threads easier to measure and compare.
- Ask students to predict the fastest thread count before conducting the experiment and then explain any difference between their prediction and the measured result. This helps challenge the common assumption that more threads always produce faster

execution.

- Allow sufficient time for discussion after the performance experiment. The measured results are most useful when students connect them to processor-core limits, overhead, synchronization, task granularity, scalability, correctness, and code complexity.

18.3.2.4 Data and Analysis

Pre- and post-activity surveys were administered to evaluate changes in students' self-reported understanding of PDC concepts and their perceptions of the Parallel Image Processing activity. After survey previews, non-consenting responses, incomplete submissions, and responses without survey-item data were excluded, 21 usable pre-activity responses and 14 usable post-activity responses remained. Because the surveys were anonymous and individual pre- and post-activity responses could not be matched, the results were analyzed descriptively rather than as paired observations.

For summary purposes, all Likert-scale responses were converted to numerical values from 1 to 7. The coding used for the three survey scales is summarized in Table 18.7.

Table 18.7: Coding of seven-point survey response scales

Scale	**Value**	**Response**
PDC understanding	1	I do not understand it at all
	2	I understand a little, but it is still confusing
	3	I have a basic idea, but I am not confident
	4	I am unsure or neutral about my understanding
	5	I mostly understand it, but I have some questions
	6	I understand it well and feel confident
	7	I completely understand it and could explain it to someone else
Interest in learning about PDC	1	Not at all interested
	2	Moderately uninterested
	3	Slightly uninterested
	4	Neutral
	5	Slightly interested
	6	Moderately interested
	7	Extremely interested
Post-activity perception	1	Strongly disagree
	2	Disagree
	3	Somewhat disagree
	4	Neutral
	5	Somewhat agree
	6	Agree
	7	Strongly agree

Positive responses for the post-activity perception items were defined as *Somewhat agree*, *Agree*, or *Strongly agree*.

Table 18.8 presents the pre- and post-activity means for the seven PDC-understanding items. The difference represents the post-activity mean minus the pre-activity mean; therefore, a positive difference indicates higher self-reported understanding after the activity.

Table 18.8: Pre- and post-activity understanding of PDC concepts in the Parallel Image Processing activity.

PDC Concept	Pre	Post	Difference
Event handling	2.57	3.07	0.50
Multithreading	2.29	3.43	1.14
Data parallelism	1.90	3.14	1.24
Race condition	1.95	3.00	1.05
Synchronization	2.14	3.29	1.14
Speedup with parallelism	1.90	3.14	1.24
Parallel overhead	2.00	2.79	0.79

Note. Understanding was measured on a seven-point scale, with higher values indicating greater self-reported understanding. Difference represents the post-activity mean minus the pre-activity mean. The usable sample sizes were $n = 21$ for the pre-activity survey and $n = 14$ for the post-activity survey.

Students reported higher post-activity understanding for all seven PDC concepts. The largest increases were observed for data parallelism and speedup with parallelism, each of which increased by 1.24 points. Multithreading and synchronization each increased by 1.14 points, while understanding of race conditions increased by 1.05 points. These gains align closely with the activity's emphasis on dividing image-processing work among multiple threads, comparing sequential and parallel execution times, and demonstrating the need for synchronization when multiple threads update shared histogram data.

Parallel overhead increased by 0.79 points, suggesting that students developed greater awareness of the costs associated with thread management, task scheduling, and coordination. Event handling increased by 0.50 points, which was the smallest gain among the seven concepts. This result is reasonable because event handling was not a primary focus of the image-processing activity.

All seven PDC concepts showed positive differences between the pre- and post-activity surveys. The post-activity means generally fell near 3, corresponding to students having a basic understanding of the concepts but not yet feeling confident. This outcome is consistent with the intended role of the activity. The Parallel Image Processing activity was a short PDC module embedded within a CS2 course rather than a complete course in parallel and distributed computing. Therefore, students were not expected to develop advanced understanding or strong confidence after the activity. Instead, the module was designed to provide initial exposure, establish foundational vocabulary, and help students connect familiar Java programming concepts to parallel execution. For example, the mean rating for data parallelism increased from 1.90, indicating little prior understanding, to 3.14, indicating that students had developed a basic understanding but still recognized the need for further learning. This level of understanding appropriately reflects the activity's role as a bridge to more advanced study in a future PDC course.

Students' mean interest in learning about PDC remained nearly unchanged, decreasing slightly from 4.76 before the activity to 4.71 afterward. This difference of -0.05 is negligible and suggests that the activity maintained students' existing level of interest rather than producing a clear overall increase. However, the post-activity responses varied considerably, indicating that students did not experience the assignment in the same way.

The post-activity survey included three additional items concerning students' experiences with the assignment. These items were not included in the pre-activity survey and therefore were analyzed only as post-activity perceptions. Table 18.9 summarizes their mean ratings and percentages of positive responses.

Table 18.9: Student perceptions of the Parallel Image Processing activity.

Survey Item	**Mean**	**Positive Responses**
I enjoyed this assignment	4.29	57.1%
I learned from this assignment	4.79	57.1%
This activity increased my interest in PDC	4.43	42.9%

Note. Responses were measured on a seven-point scale ranging from *1 = Strongly disagree* to *7 = Strongly agree*. Positive responses were defined as *Somewhat agree*, *Agree*, or *Strongly agree*. The usable post-activity sample size was $n = 14$.

The statement "I learned from this assignment" received the highest mean, at 4.79, which is close to *Somewhat agree* (5). The enjoyment item received a mean of 4.29, falling between *Neutral* (4) and *Somewhat agree* (5). The statement that the activity increased students' interest in PDC received a mean of 4.43, also falling between *Neutral* and *Somewhat agree*.

Although 57.1% of students responded positively to the statements that they enjoyed and learned from the assignment, only 42.9% agreed at least somewhat that the activity increased their interest in PDC. These results suggest that students viewed the activity more positively as a learning experience than as a source of increased interest or motivation.

Students also reported how many hours they spent on the assignment. Most interpretable responses fell between approximately three and ten hours, although the reported time varied widely. One student reported spending no time on the assignment, while another reported spending more than 24 hours. This variation suggests substantial differences in students' prior preparation, programming experience, troubleshooting needs, and ability to complete the activity independently.

Open-ended responses provided additional information about what students learned and the difficulties they encountered. The major themes are summarized in Table 18.10.

Table 18.10: Themes from open-ended Parallel Image Processing responses.

Theme	Representative Evidence	Interpretation
Multithreading and data parallelism	Students described learning how Java programs divide a problem into smaller tasks and process multiple tasks at the same time.	Students connected the programming implementation to the general idea of data-parallel execution.
Performance and optimization	Students reported learning how parallelism and multithreading can improve program performance compared with sequential processing.	The runtime experiment helped students connect parallel program design to measurable performance.
Synchronization	One student specifically identified learning about the advantages and disadvantages of synchronization.	The histogram demonstration supported understanding of race conditions, correctness, and synchronization.
Concurrency and parallelism	One response discussed comparing and contrasting parallelism and concurrency.	Students began to distinguish two closely related PDC concepts.
Programming environment	Students mentioned learning to run the code in IntelliJ and troubleshoot library or file-path problems.	Some learning focused on development-environment skills in addition to PDC concepts.
Timing and instructional difficulty	One student reported that the activity was introduced too close to the midterm, leaving insufficient time to study the new material.	In future implementations, the module will be introduced earlier in the semester, with greater separation between the activity and the midterm examination.

Overall, the survey results suggest that the Parallel Image Processing activity was associated with higher post-activity ratings of students' self-reported understanding of PDC concepts. The largest gains were observed in data parallelism, speedup with parallelism, multithreading, synchronization, and race conditions. The post-activity ratings indicate that students generally developed a basic understanding of these concepts, which aligns with the intended purpose of this short CS2 module: to introduce foundational PDC concepts and prepare students for more advanced study in a future PDC course. The open-ended responses further indicate that students learned how parallel processing divides a problem into smaller tasks, how multithreading can be implemented in Java, and how parallel execution can improve performance compared with sequential execution.

The results also suggest several opportunities for refinement. The mixed ratings for enjoyment and increased interest indicate that students' experiences with the activity varied. In addition, one student reported that the module was introduced too close to the midterm, leaving insufficient time to study the new material. In future implementations, the module will be introduced earlier in the semester, with greater separation from the midterm examination. Future pre- and post-activity surveys will also include a confidential matching identifier so that individual students' responses can be paired while preserving anonymity. This will allow changes in students' understanding and interest to be evaluated using matched statistical analyses rather than only descriptive group-level comparisons.

18.4 Course-wide Pre and Post Course Surveys

To evaluate the effectiveness of integrating parallel and distributed computing (PDC) concepts into the CS2 course, we administered pre- and post-course surveys at the beginning and end of the semester. Fall 2024 served as the baseline semester without the new teaching

modules, while Fall 2025 served as the intervention semester. Two modules were implemented during the intervention semester: the Parallel Image Processing module, which introduced multithreading and data-parallel execution, and the Greenfoot module, in which students implemented event handling by designing games. The Greenfoot work culminated in a final game-design project.

The baseline semester included 10 students with matched pre- and post-course responses. The intervention semester included 23 such students, although the matched sample size ranged from 21 to 23 for individual items because of missing responses. All pre-course means, post-course means, mean differences, and statistical analyses were calculated using the available matched responses for each item.

The survey measured students' self-reported experience with traditional CS2 programming concepts, PDC-related concepts, and attitudes toward computing. Students rated their experience with the following CS2 concepts:

1. Classes and Objects
2. Functions
3. Input/Output from/to Text Files
4. Incremental Programming
5. Arrays/Vectors
6. Sorting Algorithms with Arrays
7. Constructors
8. Access Specifiers
9. Inheritance
10. Polymorphism
11. Exceptions
12. Graphical User Interfaces (GUI)
13. Keyboard and Mouse Events
14. Recursion
15. Writing Test Cases

Students also rated their experience with the following PDC-related concepts:

1. Problem Decomposition
2. Parallel Processing
3. Distributed Computing
4. Event Handling

Finally, students responded to the following attitude and belief statements:

A1 In general, my interest in computing is high.
A2 My interest in learning about parallel and distributed computing is high.
A3 In general, I believe learning about parallel and distributed computing is important.
A4 I believe I have a good understanding of computing.
A5 I believe I have a good understanding of parallel and distributed computing.

All survey items were measured using seven-point Likert scales. The programming and PDC-related concept questions were coded from *1 = No Experience at All* to *7 = Extremely*

Experienced. Intermediate responses were coded as *Very Little Experience* = 2, *Slightly Experienced* = 3, *Between Slightly and Moderately Experienced* = 4, *Moderately Experienced* = 5, and *Very Experienced* = 6. The attitude and belief questions were coded from *1* = *Strongly Disagree* to *7* = *Strongly Agree*.

As shown in Table 18.11, students in both semesters generally reported gains in traditional CS2 programming concepts. In the baseline semester, positive gains were observed for 13 of the 15 concepts, while input/output from/to text files and constructors each decreased slightly by 0.10 points. In the intervention semester, students reported positive gains across all 15 concepts.

The intervention semester produced gains of at least one point for classes and objects, incremental programming, sorting algorithms with arrays, access specifiers, inheritance, polymorphism, exceptions, and recursion. The largest gain was observed for inheritance, which increased by 1.43 points. Exceptions increased by 1.26 points, polymorphism by 1.22 points, incremental programming by 1.17 points, and sorting algorithms with arrays by 1.16 points.

For most traditional CS2 concepts, the intervention semester showed larger gains than the baseline semester. These results suggest that incorporating the Parallel Image Processing and Greenfoot modules did not negatively affect students' perceived learning of the traditional CS2 curriculum. Instead, students continued to develop their understanding of object-oriented programming, file processing, arrays, algorithms, graphical programming, event-driven programming, recursion, testing, and related Java concepts while also receiving explicit exposure to PDC and event handling.

Table 18.11: CS2 survey results for traditional programming concepts.

	No intervention			**Intervention**		
CS2 Programming Concept	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Classes and Objects	4.10	4.70	0.60	3.91	5.00	1.09
Functions	4.40	4.90	0.50	4.04	4.87	0.83
Input/Output Text Files	4.60	4.50	-0.10	3.91	4.52	0.61
Incremental Programming	3.50	4.40	0.90	2.87	4.04	1.17
Arrays/Vectors	3.90	4.40	0.50	3.57	4.35	0.78
Sorting Algorithms with Arrays	3.90	4.30	0.40	3.36	4.52	1.16
Constructors	4.30	4.20	-0.10	3.61	4.48	0.87
Access Specifiers	3.90	4.00	0.10	3.13	4.17	1.04
Inheritance	3.80	4.80	1.00	3.30	4.74	1.43
Polymorphism	3.70	4.30	0.60	3.17	4.39	1.22
Exceptions	4.00	4.50	0.50	3.09	4.35	1.26
GUI	3.44	3.90	0.46	2.82	3.57	0.75
Keyboard and Mouse Events	3.10	3.20	0.10	2.83	3.35	0.52
Recursion	3.90	4.20	0.30	3.13	4.13	1.00
Writing Test Cases	4.20	4.30	0.10	3.48	4.13	0.65

For the PDC-related concepts, both semesters showed positive changes from the pre-course to the post-course survey, as shown in Table 18.12. In the intervention semester, the largest increase was observed for problem decomposition, which increased by 1.18 points. Distributed computing increased by 1.00 point, event handling by 0.93 points, and parallel processing by 0.52 points.

The comparison between semesters was mixed. The intervention semester showed larger gains than the baseline semester for problem decomposition, parallel processing, and event

handling, while the baseline semester showed a larger gain for distributed computing. Because the two semesters involved different groups of students with different initial experience levels, the differences in mean gains should be interpreted cautiously. Nevertheless, the positive changes across all four concepts indicate that students developed greater familiarity with PDC-related ideas during the intervention semester.

The gain in problem decomposition is relevant to both intervention modules. In the image-processing activity, students divided an image-processing problem into smaller tasks that could be executed by multiple threads. In the Greenfoot module, students decomposed a game into actors, behaviors, events, and interactions. The increase in parallel processing is directly aligned with the multithreaded image-processing activity, while the improvement in event handling is closely connected to the Greenfoot game-design module and final project. Students implemented behaviors that responded to keyboard, mouse, timing, collision, and other game events. The improvement in distributed computing suggests broader growth in students' awareness of PDC concepts beyond the specific implementations used in the two modules.

Table 18.12: CS2 survey results for PDC-related concepts.

	No intervention			**Intervention**		
PDC-Related Concept	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Problem Decomposition	4.10	4.50	0.40	3.39	4.57	1.18
Parallel Processing	2.60	3.00	0.40	2.43	2.95	0.52
Distributed Computing	1.90	3.20	1.30	1.91	2.91	1.00
Event Handling	2.60	3.40	0.80	2.39	3.32	0.93

The attitude and belief results were generally stable, as summarized in Table 18.13. Students in both semesters entered the course with relatively high interest in computing, interest in learning about PDC, and belief in the importance of learning PDC. These high initial ratings left limited room for substantial growth.

In the intervention semester, interest in computing increased slightly from 5.61 to 5.73, while interest in learning about PDC increased from 5.09 to 5.18. Students' belief in the importance of learning PDC decreased slightly from 5.74 to 5.45 but remained clearly on the positive side of the scale. Self-reported understanding of computing increased by 0.51 points, and self-reported understanding of PDC increased by 0.33 points.

Overall, the attitude results suggest that the two teaching modules maintained students' already positive interest in computing and PDC. The increase in perceived understanding of computing is also consistent with the positive gains observed across the traditional CS2 concepts. The relatively small change in perceived understanding of PDC is reasonable because the modules were designed to provide introductory exposure within CS2 rather than the depth of instruction offered by a dedicated PDC course.

Table 18.13: CS2 survey results for attitudes and beliefs.

	No intervention			Intervention		
Attitude / Belief	R_{base_pre}	R_{base_post}	$Mdiff_{base}$	R_{int_pre}	R_{int_post}	$Mdiff_{int}$
Interest in Computing	5.80	6.00	0.20	5.61	5.73	0.12
Interest in Learning PDC	5.80	6.00	0.20	5.09	5.18	0.09
Importance of Learning PDC	6.10	6.00	-0.10	5.74	5.45	-0.28
Understanding of Computing	4.80	5.10	0.30	4.26	4.77	0.51
Understanding of PDC	3.70	4.20	0.50	3.35	3.68	0.33

In addition to comparing the no-intervention and intervention semesters, we conducted a more detailed analysis of the matched pre- and post-course responses from the intervention semester. The matched responses were analyzed using the non-parametric Wilcoxon signed-rank test. Statistical significance was evaluated at $\alpha = 0.05$, and effect sizes were computed using Cliff's delta (δ). Because multiple survey outcomes were examined simultaneously, these inferential analyses are treated as exploratory, and the reported p-values were not adjusted for multiple comparisons. The results should therefore be interpreted cautiously, with consideration of both the pattern of results and the reported effect sizes.Tables 18.14–18.16 report the pre- and post-course means, mean differences, Wilcoxon statistics (W), p-values, statistical significance, Cliff's delta, and effect-size interpretations.

Table 18.14 shows positive gains across all 15 traditional CS2 concepts. Thirteen of these gains reached statistical significance at the $\alpha = .05$ level. The statistically significant results included classes and objects, functions, input/output from/to text files, incremental programming, arrays/vectors, sorting algorithms with arrays, constructors, access specifiers, inheritance, polymorphism, exceptions, GUI, and recursion.

The largest mean gain was observed for inheritance, which increased from 3.30 to 4.74, a difference of 1.43 points. Inheritance also had a large effect size. Classes and objects increased by 1.09 points and also had a large effect size. Incremental programming, sorting algorithms with arrays, polymorphism, exceptions, access specifiers, and recursion produced medium effect sizes. Although keyboard and mouse events and writing test cases did not reach statistical significance, both showed positive mean differences and small effect sizes.

These findings indicate that integrating the Parallel Image Processing and Greenfoot modules did not appear to interfere with students' self-reported understanding of traditional CS2 concepts. Students reported substantial improvement across object-oriented programming, file processing, arrays, sorting, graphical programming, event-driven programming, recursion, and testing concepts.

Table 18.14: Matched pre- and post-course survey results for traditional programming concepts in CS2.

Item	Pre	Post	Diff	W	p-value	Sig.	Cliff's δ	Interp.
Classes and Objects	3.91	5.00	1.09	0.0	$< .001$	Yes	-0.48	large
Functions	4.04	4.87	0.83	17.5	0.007	Yes	-0.43	medium
Input/Output Text Files	3.91	4.52	0.61	30.5	0.044	Yes	-0.22	small
Incremental Programming	2.87	4.04	1.17	18.0	0.002	Yes	-0.46	medium
Arrays/Vectors	3.57	4.35	0.78	15.0	0.005	Yes	-0.30	small
Sorting Algorithms with Arrays	3.36	4.52	1.16	22.5	0.005	Yes	-0.45	medium
Constructors	3.61	4.48	0.87	18.0	0.007	Yes	-0.34	medium
Access Specifiers	3.13	4.17	1.04	16.5	0.004	Yes	-0.39	medium
Inheritance	3.30	4.74	1.43	3.5	$< .001$	Yes	-0.52	large
Polymorphism	3.17	4.39	1.22	12.0	0.001	Yes	-0.42	medium
Exceptions	3.09	4.35	1.26	7.0	$< .001$	Yes	-0.47	medium
GUI	2.82	3.57	0.75	23.0	0.018	Yes	-0.38	medium
Keyboard and Mouse Events	2.83	3.35	0.52	40.5	0.078	No	-0.24	small
Recursion	3.13	4.13	1.00	32.5	0.011	Yes	-0.36	medium
Writing Test Cases	3.48	4.13	0.65	42.5	0.103	No	-0.22	small

Table 18.15 presents the matched results for the four PDC-related concepts. Students reported higher post-course ratings for all four concepts, and all four gains reached statistical significance.

Problem decomposition showed the largest mean gain, increasing from 3.39 to 4.57, with a difference of 1.18 points and a large effect size. This finding aligns with both teaching modules. The image-processing activity required students to divide image-processing work among multiple threads, while the Greenfoot project required students to decompose a game into interacting classes, actors, behaviors, and events.

Distributed computing increased by 1.00 point, and event handling increased by 0.93 points; both produced medium effect sizes. The improvement in event handling is particularly relevant to the Greenfoot module, in which students designed a game and implemented behaviors triggered by user input, collisions, timing, and changes in the game state. Parallel processing increased from 2.43 to 2.95, a difference of 0.52 points and a small effect size. This gain is directly related to the Parallel Image Processing module, which introduced students to thread-based task partitioning, synchronization, overhead, and speedup.

The post-course ratings indicate that students developed foundational experience with these concepts rather than advanced mastery. This outcome is consistent with the purpose of the two modules, which was to introduce PDC and event-driven programming within CS2 and provide a bridge to more advanced study.

Table 18.15: Matched pre- and post-course survey results for PDC-related concepts in CS2.

Item	Pre	Post	Diff	W	p-value	Sig.	Cliff's δ	Interp.
Problem Decomposition	3.39	4.57	1.18	9.0	0.001	Yes	-0.50	large
Parallel Processing	2.43	2.95	0.52	35.0	0.042	Yes	-0.27	small
Distributed Computing	1.91	2.91	1.00	15.0	0.009	Yes	-0.40	medium
Event Handling	2.39	3.32	0.93	36.5	0.016	Yes	-0.41	medium

Table 18.16 presents the matched results for students' attitudes and perceptions. Students

entered the course with relatively high interest in computing, interest in learning about PDC, and belief in the importance of learning PDC. Consequently, there was limited room for substantial improvement in these areas.

Interest in computing increased slightly from 5.61 to 5.73, while interest in learning about PDC increased from 5.09 to 5.18. Both changes had negligible effect sizes. Students' belief in the importance of learning PDC decreased slightly from 5.74 to 5.45, but the post-course mean remained above the neutral midpoint and continued to indicate general agreement with the importance of PDC.

Students' perceived understanding of computing increased from 4.26 to 4.77, a difference of 0.51 points and a small effect size. Their perceived understanding of PDC increased from 3.35 to 3.68, a difference of 0.33 points and a negligible effect size. None of the five attitude and perception changes reached statistical significance. Overall, the results suggest that the modules preserved students' positive attitudes toward computing and PDC while producing modest increases in their perceived understanding.

Table 18.16: Matched pre- and post-course survey results for attitudes and perceptions in CS2.

Item	**Pre**	**Post**	**Diff**	W	p**-value**	**Sig.**	**Cliff's** δ	**Interp.**
Interest in Computing	5.61	5.73	0.12	13.5	0.248	No	-0.08	negligible
Interest in Learning PDC	5.09	5.18	0.09	25.0	0.449	No	-0.07	negligible
Importance of Learning PDC	5.74	5.45	-0.28	28.0	0.372	No	0.09	negligible
Understanding of Computing	4.26	4.77	0.51	29.0	0.068	No	-0.15	small
Understanding of PDC	3.35	3.68	0.33	35.0	0.147	No	-0.14	negligible

Note. Responses were coded on seven-point Likert scales. Diff represents the post-course mean minus the pre-course mean. Wilcoxon signed-rank tests were two-sided, with statistical significance evaluated at $\alpha = .05$. Cliff's δ was calculated as $\delta(\text{pre}, \text{post})$; therefore, negative values indicate that post-course ratings tended to be higher than pre-course ratings. The matched sample size was $n = 23$ for most traditional CS2 items, $n = 22$ for sorting algorithms with arrays and GUI, $n = 21$ for problem decomposition, and $n = 22$ for the remaining PDC-related and attitude items.

The baseline semester included only 10 matched responses, while the intervention semester included between 21 and 23 matched responses depending on the survey item. Therefore, the results should be interpreted cautiously. The two semesters also involved different student cohorts with different pre-course experience levels, so differences in gains cannot be attributed solely to the Parallel Image Processing and Greenfoot modules.

Nevertheless, the results provide encouraging evidence that the two modules were integrated without reducing students' progress in the traditional CS2 curriculum. Students in the intervention semester reported positive gains across all 15 traditional CS2 concepts, with statistically significant improvement in 13 of them. They also reported statistically significant gains in problem decomposition, parallel processing, distributed computing, and event handling. The results are consistent with the complementary purposes of the two modules: the Parallel Image Processing module introduced multithreading and data-parallel execution, while the Greenfoot game-design module provided practical experience with event handling and event-driven programming. Together, the modules offered foundational exposure to PDC-related ideas while preserving and reinforcing students' learning of core CS2 material.

18.5 Other Activities and Ideas

No additional PDC activities were added to CS2 because the Parallel Image Processing and Greenfoot Event Handling activities already required three 85-minute class sessions. These activities provided students with practical experience in both data-parallel and event-driven programming while preserving sufficient class time for the traditional CS2 curriculum.

18.6 Conclusion

The CS2 integration shows that PDC concepts can be incorporated into a CS2 course without replacing the core Java programming curriculum. The course retained its focus on object-oriented programming, GUI programming, recursion, collections, and standard CS2 problem solving, while adding two applied learning modules: Image Processing for Data Parallelism and the Greenfoot Project for Event Handling.

The image-processing module gave students experience with data parallelism, Java multithreading, synchronization, speedup, and performance analysis in a visual and practical context. The Greenfoot project served as a motivating final project in which students applied event handling along with broader CS2 concepts such as object-oriented design, class interaction, conditionals, loops, debugging, documentation, teamwork, and presentation.

Survey results showed gains in students' self-reported understanding of PDC-related concepts, and project performance suggested that most students were able to complete the applied event-handling project successfully. Overall, the CS2 experience suggests that carefully selected programming activities can help students connect CS2 content to modern parallel, concurrent, and event-driven computing practices.

Appendix

Github Link to Instructional Material

https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS2/chapter18-MSU

Detailed Pre and Post Syllabus

The Parallel and Distributed Computing (PDC) intervention was implemented in CSIT 112, Fundamentals of Programming II, the CS2 course at Montclair State University. CSIT 112 is a Java-based second programming course covering core CS2 topics such as review of introductory programming, polymorphism, exception handling, recursion, collections, and event handling. The Fall 2024 semester served as the baseline semester before the PDC intervention was introduced. The Fall 2025 semester incorporated two plugged activities: a parallel image processing activity and an event-handling application using Greenfoot.

The pre-intervention syllabus followed the standard CS2 structure and focused on object-oriented programming, recursion, collections, exception handling, and related Java programming topics without an explicit PDC image-processing module. The post-intervention syllabus preserved the existing CS2 course structure while integrating PDC concepts through the image processing activity and event-driven programming through the Greenfoot project. The image processing activity was implemented as two 85-minute class meetings: one class for lecture and sample-code demonstration, and one class for the lab. Through this activity, students explored data parallelism, multithreading, speedup, scalability, synchronization, load balancing, and performance evaluation. The Greenfoot event-handling activity included approximately 30 minutes of event-handling slides and approximately 20 minutes for project introduction and sample project demonstration. Students then applied event-driven programming concepts by designing interactive games that respond to mouse and keyboard input. The pre- and post-course experience surveys were collected during the baseline and intervention semesters to compare student experience, attitudes, and PDC-related learning outcomes. In this way, the PDC intervention was added to an otherwise standard CS2 sequence while preserving the existing core course content.

Organization of Material

The `chapter18-MSU` directory is organized as follows:

```
chapter18-MSU/
COURSE_CALENDAR.md
Detailed_Pre_and_Post_Syllabus/
Evaluation_Data_and_Analysis/
Plugged_Activities/
  Image_Processing/
  Greenfoot_Event_Handling/
Unplugged_Activities/
```

`README.md`

`COURSE_CALENDAR.md`

Purpose: This file (available here) provides the Fall 2025 course calendar for CSIT 112 at Montclair State University.

Contents: A week-by-week schedule of course modules, class meetings, labs, assignments, examinations, breaks, and the two plugged activities: Parallel Image Processing and Greenfoot Event Handling.

Usage: Primary reference for understanding where the PDC and event-handling activities were placed within the overall CS2 course sequence.

`Detailed_Pre_and_Post_Syllabus`

Purpose: This directory (available here) contains the pre- and post-intervention syllabus materials.

Contents: The Fall 2024 pre-PDC syllabus and the Fall 2025 post-PDC syllabus for CSIT 112.

Usage: Reference materials for comparing the baseline CS2 course structure with the semester in which the PDC and Greenfoot activities were implemented.

`Evaluation_Data_and_Analysis`

Purpose: This directory (available here) contains the anonymized and aggregate evaluation materials used to assess the intervention.

Contents: Aggregate analysis outputs comparing the Fall 2024 baseline semester and the Fall 2025 intervention semester, including student programming experience, attitudes and beliefs, and PDC-related survey results.

Usage: Source for evaluation summaries, comparison tables, and analysis of student learning and experience before and after the CS2 intervention.

`Plugged_Activities`

Purpose: This directory (available here) contains the plugged instructional activities used in the CS2 course package.

Contents: The directory includes two activity folders: `Image_Processing`, which contains the parallel image processing lecture, sample code, lab handout, and lab code; and `Greenfoot_Event_Handling`, which contains the event-handling slides, Greenfoot project instructions, and sample project.

Usage: Classroom instruction materials for introducing PDC concepts through image processing and event-driven programming through Greenfoot.

`Unplugged_Activities`

Purpose: This directory (available here) documents that no unplugged activity was implemented as part of this CS2 course package.

Contents: A short README noting that the CS2 intervention used plugged activities rather than unplugged activities.

Usage: Clarifies the activity structure for readers and directs them to the plugged activity materials.

`README.md`

Purpose: Quick reference documentation for the MSU CS2 course package.

Contents: Overview of the course package, activity summary, and repository organization.

Usage: Entry point for users to understand the module's organization and purpose.

Survey and Other Anonymized Data and Analysis

On the GitHub link, the directory `Evaluation_Data_and_Analysis` contains anonymized aggregate analysis files for evaluating the CS2 intervention. The Fall 2024 semester served as the baseline semester, and the Fall 2025 semester included the Parallel Image Processing and Greenfoot Event Handling activities. The analysis materials include comparison outputs for student programming experience, attitudes and beliefs, and PDC-related learning outcomes. Raw student-level survey files and open-ended responses are not included in the public repository in order to protect student privacy.

Bibliography

[1] Lorin W. Anderson and David R. Krathwohl, editors. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, New York, NY, 2001.

[2] Lorin W Anderson and David R Krathwohl. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives: complete edition*. Addison Wesley Longman, Inc., 2001.

[3] OpenMP Architecture Review Board. Openmp application programming interface (api) specification, version 6.0. Technical report, OpenMP Architecture Review Board, November 2024. URL https://www.openmp.org/specifications/. Online. Available: https://www.openmp.org/specifications/.

[4] Steven A Bogaerts. Limited time and experience: Parallelism in cs1. In *2014 ieee international parallel & distributed processing symposium workshops*, pages 1071–1078. IEEE, 2014.

[5] Chris Bourke. Codeless modules for parallel and distributed computing in early computing curriculum. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1*, pages 141–147, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722561. doi: 10.1145/3770762.3772500. URL https://doi.org/10.1145/3770762.3772500.

[6] Chris Bourke and Justin Firestone. Codeless PDC modules for early computing curriculum. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 357–364, 2024. doi: 10.1109/IPDPSW63119.2024.00082.

[7] Steven Bradley, Miranda C. Parker, Rukiye Altin, Lecia Barker, Sara Hooshangi, Thom Kunkeler, Ruth G. Lennon, Fiona McNeill, Julià Minguillón, Jack Parkinson, Svetlana Peltsverger, and Naaz Sibia. Modeling women's elective choices in computing. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE–WGR '23, page 196–226, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704055. doi: 10.1145/3623762.3633497. URL https://doi.org/10.1145/3623762.3633497.

[8] Neil C. C. Brown. Bluej 5: Still going strong. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1420, New York, NY,

USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3372542. URL https://doi.org/10.1145/3328778.3372542.

[9] H. Martin Bucker, Johannes Schoder, Xiaoyuan Suo, and D. Bunde. Peachy parallel assignments. In *Proceedings of EduPar-25: 15th NSF/TCPP Workshop on Parallel and Distributed Computing Education*, Milan, Italy, June 2025. Held in conjunction with the 39th IEEE International Parallel & Distributed Processing Symposium (IPDPS).

[10] David P. Bunde. Chapter 4. In Sushil K. Prasad, Anshul Gupta, Arnold L. Rosenberg, Alan Sussman, and Charles Weems, editors, *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Computer Science Courses*. Morgan Kaufmann, 1st edition, 2018. ISBN 9780128038994.

[11] D.P. Bunde. Modules for introducing threads. In S.K. Prasad, A. Gupta, A.L. Rosenberg, A. Sussman, and C.C. Weems, editors, *Topics in parallel and distributed computing: Introducing concurrency in undergraduate courses*, chapter 4, pages 59–82. Morgan Kaufmann, 2015.

[12] D.P. Bunde, A.R. Crockett, G.C. Gannod, J. Spacco, N. Thota, and C.C. Weems. Wip: Updating CS1 to a 21st-century model of computing. In *Proc. Frontiers in Education (FIE)*, 2024.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 4th edition, 2022.

[14] April R. Crockett. Self-selected experience-based grouping in CS1: Examining student success and persistence in CS major. pages 1742–1742. ACM, 2 2026. ISBN 9798400722554. doi: 10.1145/3770761.3777061. URL https://dl.acm.org/doi/10.1145/3770761.3777061.

[15] April R. Crockett and Gerald C. Gannod. Beyond CS1: Longitudinal effects of student-selected experience-based grouping on success and persistence. In *2026 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. Institute of Electrical and Electronics Engineers (IEEE), 2 2026.

[16] April R. Crockett, Gerald C. Gannod, and Moumita Kamal. Improving student success and retention in CS1 through self-selection into experience-based groups. In *2023 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 10 2023. ISBN 979-8-3503-3642-9. doi: 10.1109/FIE58773.2023.10342950. URL https://ieeexplore.ieee.org/document/10342950/.

[17] April R. Crockett, Gerald C. Gannod, and Neena Thota. Enhancing student success in CS1: A self-selected experience-based grouping approach. In *2025 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. Institute of Electrical and Electronics Engineers (IEEE), 1 2025. doi: 10.1109/fie63693.2025.11328570. URL https://ieeexplore.ieee.org/abstract/document/11328570.

[18] April R. Crockett, Gerald C. Gannod, Xiaoyuan Suo, Chris Bourke, Mary L. Smith, Srishti Srivastava, David P. Bunde, Jaime Spacco, Jiayin Wang, Michelle Zhu, Neena Thota, Charles C. Weems, R. Vaidyanathan, Alan Sussman, and Sushil K. Prasad. Making room for parallel and distributed computing in CS1: approaches, tradeoffs, and faculty effort. In *2026 Frontiers in Education (FIE), Accepted*, pages –. IEEE, 2026. URL https://doi.org/10.1109/FIE61694.2024.10893534.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[20] Educational Content Team. Convergent thinking. Cogn-IQ Encyclopedia, 2026. URL https://pubscience.org/cqep.2025.0105.

[21] Fawzi Emad. Flag coloring programming assignment. Personal communication to J. Spacco. URL https://www.cs.umd.edu/people/fpe.

[22] Tony Gaddis. *Starting Out with C++: From Control Structures through Objects*. Pearson, 10th edition, 2021. ISBN 9780135235003. Revel Edition.

[23] Tony Gaddis. *Starting Out with C++ from Control Structures to Objects*. Pearson, 10th edition, 2023. ISBN 9780137450626. Pearson eText platform.

[24] M. Gerten, J.M. Iqrah, and J. Spacco. Knoxcraft Terps: Parallel Minecraft for novices, a Peachy Assignment. In *Proc. 16th NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar)*, 2026.

[25] M. Graf and D.P. Bunde. Using wrappers to simplify task parallel programming. In *Proc. 21st Annual Consortium for Computing Science in Colleges Midwestern conference*, pages 73–79, 2014.

[26] D. Grossman. Sophomoric parallelism and concurrency. https://homes.cs.washington.edu/~djg/teachingMaterials/spac/, viewed Aug 18, 2026.

[27] D. Grossman. Ready-for-use: 3 weeks of parallelism and concurrency in a required second-year data-structures course. In *Proc. Workshop on Curricula for Concurrency and Parallelism*, 2010.

[28] D. Grossman and R.E. Anderson. Introducing parallelism and concurrency in the data structures course. In *Proc. 43rd SIGCSE Technical Symp. Computer Science Education (SIGCSE TS)*, pages 505–510, 2012. doi: https://doi.org/10.1145/2157136.2157285.

[29] Joy Paul Guilford. The structure of intellect. *Psychological Bulletin*, 53(4):267–293, 1956. doi: 10.1037/h0040755.

[30] NSF/IEEE-TCPP Curriculum Initiative. Peachy parallel assignments. https://tcpp.cs.gsu.edu/curriculum/?q=peachy, viewed July 27, 2026.

[31] David H. Jonassen. Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4):63–85, 2000. doi: 10.1007/BF02300500.

[32] Andrew T Kitchen, Nan C Schaller, and Paul T Tymann. Game playing as a technique for teaching parallel computing concepts. *ACM SIGCSE Bulletin*, 24(3):35–38, 1992.

[33] Michael Kölling. The Greenfoot programming environment. *ACM Trans. Comput. Educ.*, 10(4), November 2010. doi: 10.1145/1868358.1868361. URL https://doi.org/10.1145/1868358.1868361.

[34] Michael Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (ToCE)*, 10(4), 2010.

[35] Amruth N Kumar, Rajendra K Raj, Sherif G Aly, Monica D Anderson, Brett A Becker, Richard L Blumenthal, Eric Eaton, Susan L Epstein, Michael Goldweber, Pankaj Jalote, et al. Computer science curricula 2023, 2024.

[36] Chase Lambert. Makefile Tutorial by Example. Technical report, Makefile Tutorial, July 2026. URL https://makefiletutorial.com/. Online; available at https://makefiletutorial.com/.

[37] John Lewis and William Loftus. *Revel for Java Software Solutions: Foundations of Program Design*. Pearson, 10th edition, 2025. ISBN 9780138079529. REVEL Access Code Card.

[38] C. Lin and L. Snyder. *Principles of parallel programming*. Pearson Addison-Wesley, 2008.

[39] Niels Lohmann. JSON for Modern C++. Technical report, nlohmann/json, July 2025. URL https://json.nlohmann.me/. Online. Available: https://json.nlohmann.me/.

[40] Suzanne J. Matthews. Pdcunplugged: A free repository of unplugged parallel distributed computing activities. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 284–291, 2020. doi: 10.1109/IPDPSW50202.2020.00060.

[41] Suzanne J Matthews. Pdcunplugged: A free repository of unplugged parallel distributed computing activities. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 284–291. IEEE, 2020.

[42] David Olsen and Mark J. Nelson. The narrative logic of rube goldberg machines. In Nuno Nunes, Ian Oakley, and Valentina Nisi, editors, *Interactive Storytelling*, pages 104–116, Cham, 2017. Springer International Publishing. ISBN 978-3-319-71027-3.

[43] O. Ozturk, B. Glick, J. Mache, and D.P. Bunde. Peachy parallel assignments (EduPar 2019). In *Proc. 9th NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2019.

[44] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[45] N. Parlante. String-2.doubleChar. https://codingbat.com/prob/p165312, 2017, viewed Oct 2026.

[46] J. Pearce. Backtracking and caves. https://engage-csedu.org/find-resources/backtracking-and-caves, viewed Sept 2026.

[47] L. Porter, C.B. Lee, and B. Simon. Halving fail rates using peer instruction: a study of four computer science courses. In *Proc. 44th SIGCSE Technical Symp. Computer Science Education (SIGCSE TS)*, pages 177–182, 2013.

[48] L. Porter, D. Zingaro, C. Lee, C. Taylor, K.C. Webb, and M. Clancy. Developing course-level learning goals for basic data structures in CS2. In *Proc. 49th SIGCSE Technical Symp. Computer Science Education (SIGCSE TS)*, pages 858–863, 2018.

[49] Leo Porter and Beth Simon. Retaining nearly one-third more majors with a trio of instructional best practices in CS1. In *Proc. 44th SIGCSE Technical Symp. Computer Science Education (SIGCSE TS)*, pages 165–170, 2013.

[50] Sushil Prasad, Charles Weems, Alan Sussman, Anshul Gupta, Trilce Estrada, Ramachandran Vaidyanathan, Sheikh Ghafoor, Krishna Kant, and Craig Stunkel. Nsf/ieee-tcpp curriculum on parallel and distributed computing for undergraduates - version ii - big data, energy, and distributed computing. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2023, page 1220–1221, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394338. doi: 10.1145/3545947.3569594. URL https://doi.org/10.1145/3545947.3569594.

[51] Sushil K Prasad, Almadena Chtchelkanova, Sajal K Das, Frank Dehne, Mohamed G Gouda, Anshul Gupta, Joseph Jaja, Krishna Kant, Anita La Salle, Richard LeBlanc, et al. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: Core topics for undergraduates. In *SIGCSE*, volume 11, pages 617–618, 2011. URL https://dl.acm.org/doi/pdf/10.1145/1953163.1953336.

[52] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11): 60–67, November 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL https://doi.org/10.1145/1592761.1592779.

[53] Susan Rodger. Learning automata and formal languages interactively with jflap. *SIGCSE Bull.*, 38(3):360, jun 2006. ISSN 0097–8418. doi: 10.1145/1140123.1140270. URL https://doi.org/10.1145/1140123.1140270.

[54] Brandon Rodriguez, Stephen Kennicutt, Cyndi Rader, and Tracy Camp. Assessing computational thinking in cs unplugged activities. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 501–506. Association for Computing Machinery, 2017. ISBN 9781450346986. doi: 10.1145/3017680.3017779.

[55] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys. *Presented at the annual meeting of the Florida Association of Institutional Research*, 177, 2006.

[56] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. Algorithm visualization: The state of the field. *ACM Trans. Comput. Educ.*, 10(3), aug 2010. doi: 10.1145/1821996.1821997. URL https://doi.org/10.1145/1821996.1821997.

[57] M. Smith, S. Srivastava, D.P. Bunde, A. Crockett, M. Gerten, P. Maher, J. Spacco, X. Suo, J. Wang, and M. Zhu. A visual unplugged activity to introduce PDC. In *Proc. Workshop on Education for High-Performance Computing (EduHPC)*, 2025.

[58] Mary Smith, Srishti Srivastava, David P Bunde, April Crockett, Michael Gerten, Peter Maher, Jaime Spacco, Xiaoyuan Suo, Jiayin Wang, and Michelle Zhu. A visual unplugged activity to introduce pdc. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 658–665. IEEE, 2025.

[59] Srishti Srivastava, Mary Smith, Amrita Ghimire, and Shaun Gao. Assessing the integration of parallel and distributed computing in early undergraduate computer science curriculum using unplugged activities. In *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, pages 17–24. IEEE, 2019.

[60] Srishti Srivastava, Mary Smith, Amrita Ghimire, and Shaun Gao. Assessing the integration of parallel and distributed computing in early undergraduate computer science curriculum using unplugged activities. In *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, pages 17–24, 2019. doi: 10.1109/EduHPC49559.2019.00008.

[61] Daniel Stenberg and curl contributors. curl. Technical report, The curl project, July 2026. URL https://curl.se/. Online; available at https://curl.se/.

[62] Xiaoyuan Suo and Timila Dangol. *Engaging First and Second Year Undergraduates with Parallel and Distributed Computing: Lessons from Unplugged and Game-Based Activities*, pages 104–114. 06 2026. ISBN 978-3-032-22201-5. doi: 10.1007/978-3-032-22202-2_9.

[63] Xiaoyuan Suo, Olga Glebova, David Liu, Alina Lazar, and Doina Bein. A survey of teaching pdc content in undergraduate curriculum. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1306–1312. IEEE, 2021.

[64] Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide. A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Trans. Comput. Educ.*, 9(2), jun 2009. doi: 10.1145/1538234.1538236. URL https://doi.org/10.1145/1538234.1538236.

[65] Wikipedia user Zscout370. Flag of mauritius. https://en.wikipedia.org/wiki/Flag_of_Mauritius#/media/File:Flag_of_Mauritius.svg, viewed Jun 19, 2026.

[66] Frank Vahid and Roman Lysecky. Programming in C++. https://www.zybooks.com/catalog/programming-c-plus-plus/, 09 2025. Interactive online textbook.

[67] Tamar Vilner, Ela Zur, and Shay Tavor. Integrating Greenfoot into CS1: a case study. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, page 350, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306973. doi: 10.1145/1999747.1999864. URL https://doi.org/10.1145/1999747.1999864.

[68] Jiayin Wang, Michelle Zhu, Mary Smith, and Srishti Srivastava. Seeing concurrency: A java-based image processing approach to parallel computing. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.2*, SIGCSE TS 2026, page 1577–1578, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722554. doi: 10.1145/3770761.3777211. URL https://doi.org/10.1145/3770761.3777211.

[69] Benjamin L. Wiggins, Sarah L. Eddy, Lauren Wener-Fligner, Katherine Freisem, Daniel Z. Grunspan, Elli J. Theobald, Jessica Timbrook, and Alison J. Crowe. ASPECT: A Survey to Assess Student Perspective of Engagement in an Active-Learning Classroom. *CBE—Life Sciences Education*, 16(2):ar32, Summer 2017. doi: 10.1187/cbe.16-08-0244.

[70] Benjamin L. Wiggins, Sarah L. Eddy, Lauren Wener-Fligner, Katherine Freisem, Daniel Z. Grunspan, Elli J. Theobald, Jessica Timbrook, and Alison J. Crowe. ASPECT: A Survey to Assess Student Perspective of Engagement in an Active-Learning Classroom. *CBE—Life Sciences Education*, 16(2):ar32, Summer 2017. doi: 10.1187/cbe.16-08-0244.

[71] Wikipedia. Cartographers (board game). https://en.wikipedia.org/wiki/Cartographers_(board_game), viewed Sept 2026.