CDER eBOOK • Aug 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/

August 5, 2026

[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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts.* CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Contents

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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 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 first release focuses on the CS1 portion of the e-book and is being made available early to support possible Fall-term adoption. It includes eight institutional CS1 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; 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 topics rather than treated only as additional content.

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

[1] NSF Exemplar Project Award #2321015

set of common activities described in Chapter 1 and then consult institutional examples for implementation details in Chapters 2–9, each with their comprehensive 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. CS2 chapters are expected soon for subsequent-term adoption. 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 relevant chapters as indicated in that section.
- 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 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.

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, were very effective. 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

[3] NSF Institute Conceptualization Award #2002649

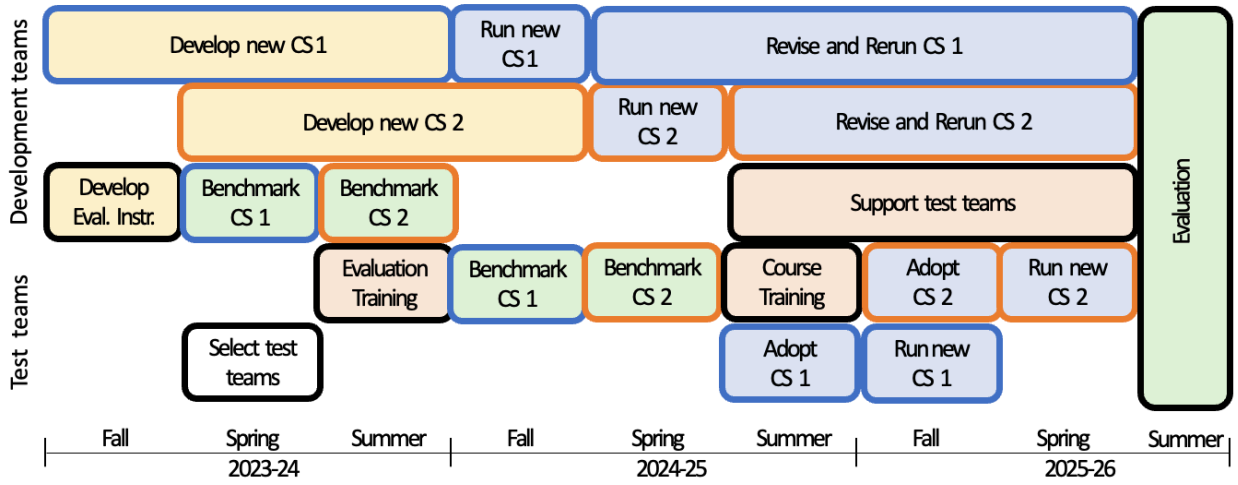


Figure 1: Project activities and Timeline. Box fill colors categorize activities as development (yellow), evaluation (light green), training and support (peach), and running the new courses (purple).

flag-grid cells in sequential and parallel scenarios.

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

- It describes, in eight chapters, various ways of infusing PDC in CS1. These chapters are complete with instructional material, classroom-based evaluation, and adoption and adaptation experiences.
- It distills a set of popular unplugged and programming activities alongside infusion methods and challenges faced into Chapter 1.
- 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 activity chapter, the institutional CS1 chapters, appendices, and repositories. The appendix 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 [11] and experience with CS1 informed our approach to CS2. The main ideas introduced can be summarized as follows:

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

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 first 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 the same PDC concepts were covered, 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 (currently 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++

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

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 lead 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.

Students:

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.

Faculty Instructors - Development and Test Teams

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, who 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 to understand the common CS1 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 CS1 infusion. Figure 2 summarizes three common reading paths.

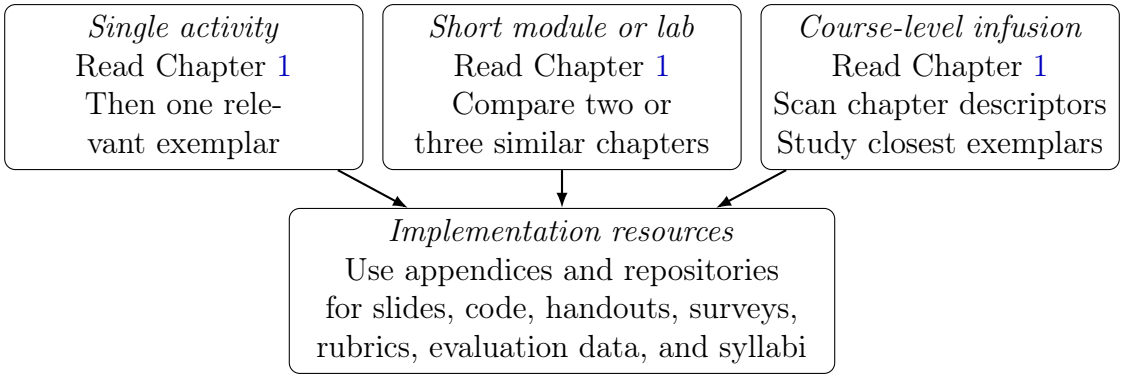


Figure 2: Suggested reading paths by CS1 adoption goal.

For a single class activity, begin with Chapter 1 and look for unplugged, visual, or short discussion-based activities. For a short module or lab, read the common description in Chapter 1 and compare two or three institutional chapters with similar constraints. For a course-level infusion, browse through Chapter 1 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.

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 CS1 module.

0.5.2 CS1 Exemplars at a Glance

Table 1 gives a compact view of the eight institutional CS1 exemplars. Readers should

Table 1: Quick comparison of CS1 institutional exemplars for potential adopters.

Chapter	Institutional context	Language	Primary PDC infusion style	Good starting point for
Ch. 2 TNTECH	Mid-sized, public university; 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. 3 Knox	Small residential liberal arts college; 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. 4 USI	Public master's-level institution; introductory object-oriented CS1 course	Java	Unplugged Flag Maker Activity, Unplugged Penny Activity, Plugged-In Flag Maker Activity, Plugged-In Parallel Sort Penny Activity, Greenfoot Activity, and activity-level assessment	Structured activity adoption; evidence-rich unplugged and plugged-in examples
Ch. 5 UNL	Large R1 institution; multiple CS1 sections and computing majors	C, Java	Codeless modules, visual/conceptual activities, remote-data activities; low and no-code pathway; large-institution adaptation; easy online, asynchronous, or hybrid adoption	Large enrollment course, online/asynchronous/hybrid courses, low-code entry introduction
Ch. 6 Webster	Private primarily undergraduate institution; small-to-moderate CS1 sections	C++	Animations, visualizations, performance examples, and game/simulation-oriented activities	Visualization-first adoption; low-barrier conceptual entry points
Ch. 7 Casper	Two-year public community college; 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. 8 HPU	Private undergraduate-serving institution with 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
Ch. 9 MSU	Public university; focused CS1 intervention	Java	Flag Maker-centered data-parallelism activity with supporting animations and assessment	Minimal-infusion model; short, focused CS1 intervention

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 Chapter 1 with Institutional Chapters

Chapter 1 provides the common entry point for CS1 activities that appear across multiple institutions. It distills recurring activity families, shared PDC concepts, common implementation challenges, and cross-institution lessons learned. Most readers should begin there, especially if looking for activities such as Flag Maker, Penny Search or Penny Sorting, animations, visualizations, remote-data/API work, event-driven examples, or lightweight programming assignments. Figure 3 shows how Chapter 1 is organized.

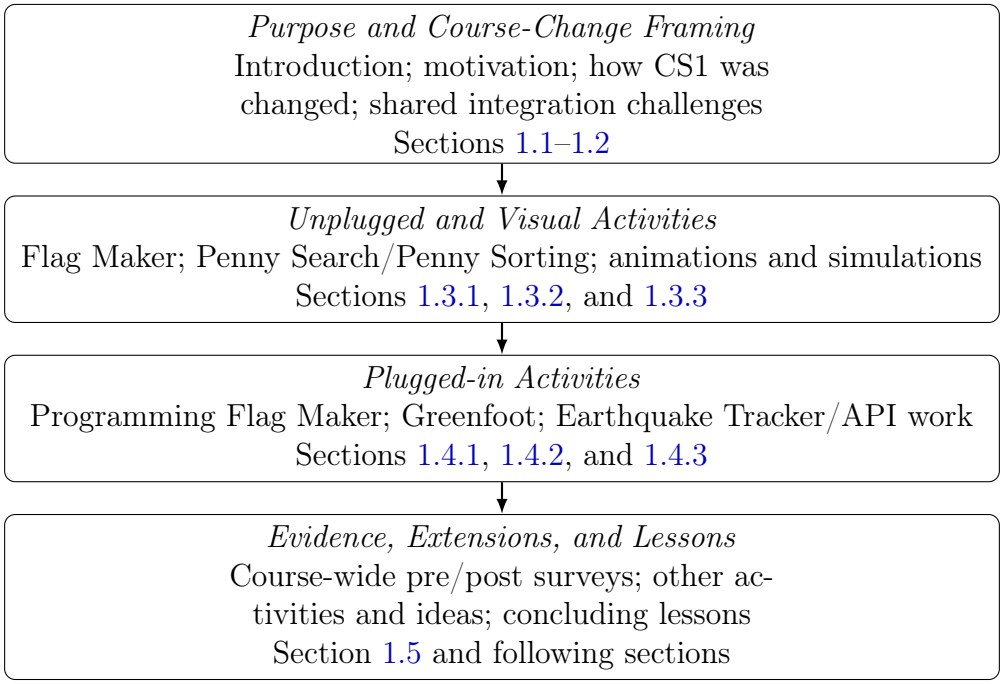


Figure 3: Organization of Chapter 1 – common CS1 activities and adoption-guidance.

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

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. Before using a resource, adopters should verify whether software versions, API links, datasets, and institutional assessment requirements remain current.

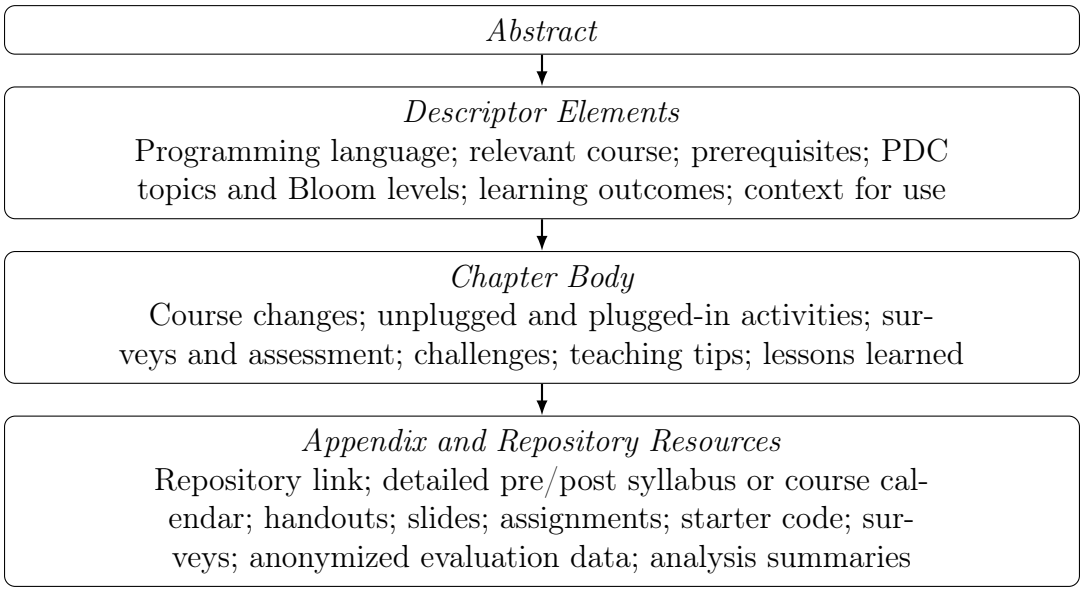


Figure 4: Common structure of institutional exemplar chapters.

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 chapters. 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. 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

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 to understand the common activity pattern, 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 CS1 modernization.

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 to 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 [39], and Version IIβ was released in 2020 with expanded attention to Big Data, energy, distributed computing, and pervasive topics [38].[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

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

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 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, Award #2321015, 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 workshops 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.

Concluding remarks

Taken together, these activities created the foundation for the present CS1/CS2 exemplar project. The TCPP curriculum initiative identified what PDC concepts undergraduates should learn; early adopters and training workshops showed both the promise and difficulty of incorporating those concepts into real courses; the CDER book series and courseware repository provided mechanisms for dissemination; and NSF-supported planning and implementation grants clarified the need for transferable, evidence-informed exemplars. The work reported in this volume is therefore the next step in a longer CDER effort: moving from guidelines, isolated modules, and instructor training toward complete, classroom-tested introductory course exemplars that can help departments modernize the computational model students encounter at the beginning of their computing education.

Acknowledgements

This work was supported in part by the National Science Foundation through Award #2321015, *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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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 Flag Maker section.
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.

Continued on next page.

Table 1.1 continued.

Activity and brief description	Primary PDC ideas	CS1 anchors	Duration / where used
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.
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.
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 [16]. 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 [19]. The programming activity is described in §1.4.1.

This activity has been previously published [45]; the material on it here draws from that document.

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:

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.

Figure 1.1: Flag of Mauritius [53]

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.

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.

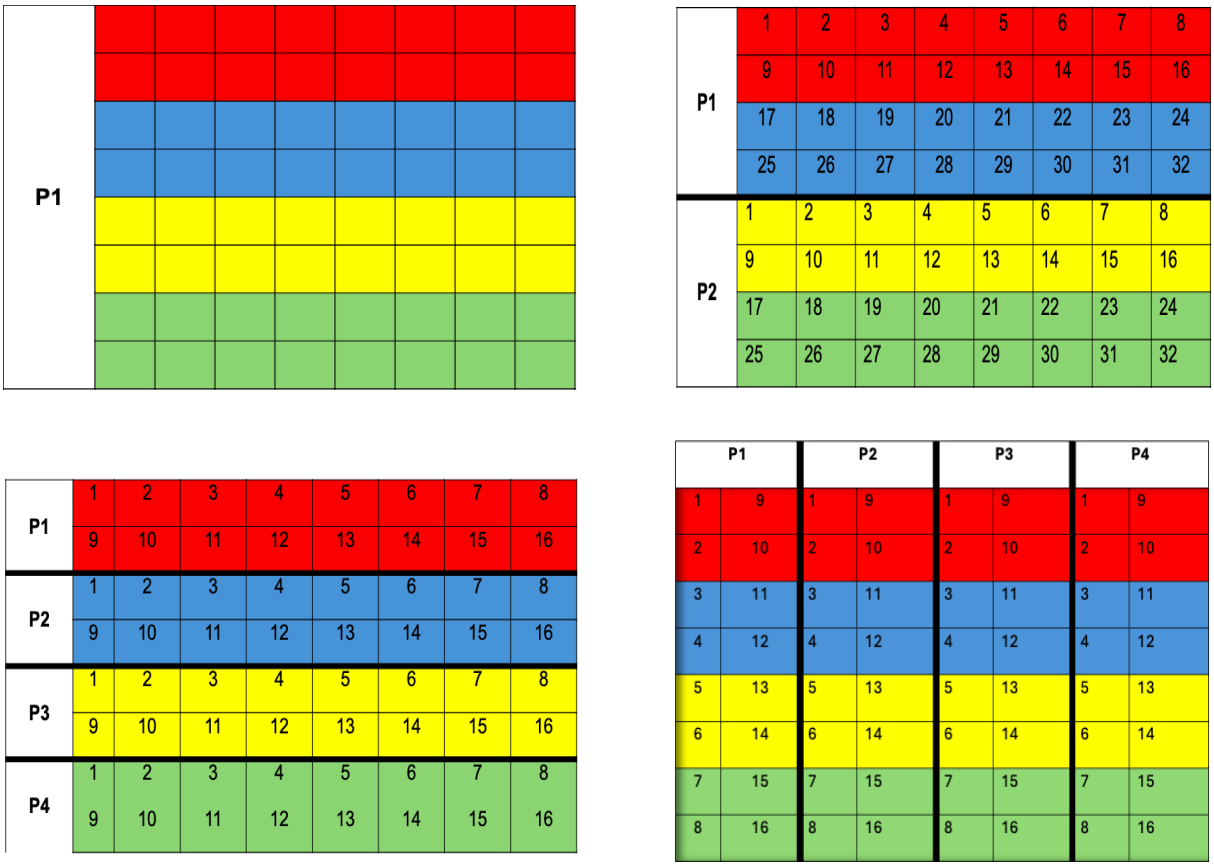


Figure 1.2: Graphical depiction of the Flag Maker scenarios: Scenario 1 (top left), Scenario 2 (top right), Scenario 3 (bottom left), and Scenario 4 (bottom right). P1–P4 depict the regions to be colored by each "processor". The cells after Scenario 1 are numbered to show the order in which they should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

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

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 joint initiative mandated by the National Science Foundation (NSF) and the ACM/IEEE Computer Society. Pointing students to the PDC curriculum rationale (available at `CDERcenter.org`) 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 significantly increases 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 [57]) 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 [47]. 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 [57]. 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

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

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 [47]. 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.

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

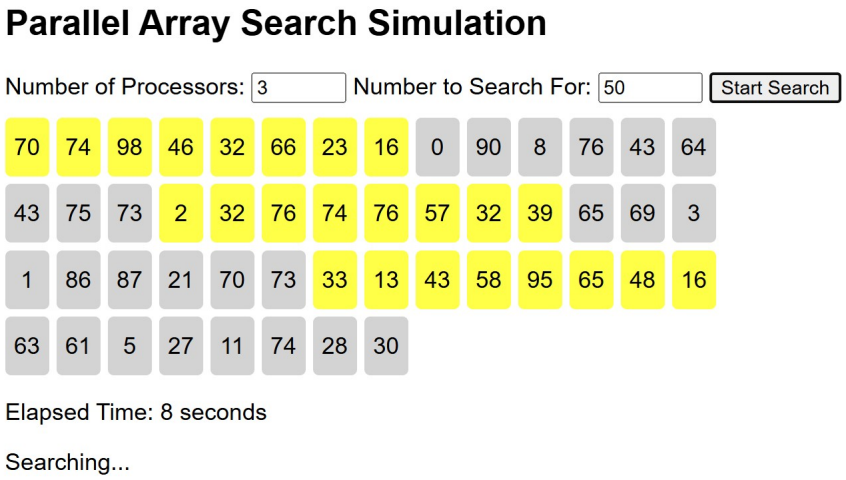


Figure 1.4: Parallel Search Animation

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.

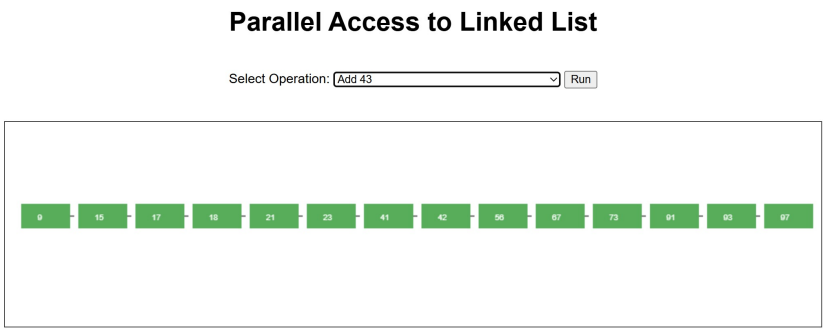


Figure 1.5: Parallel Linked List Animation

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 contiguous 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

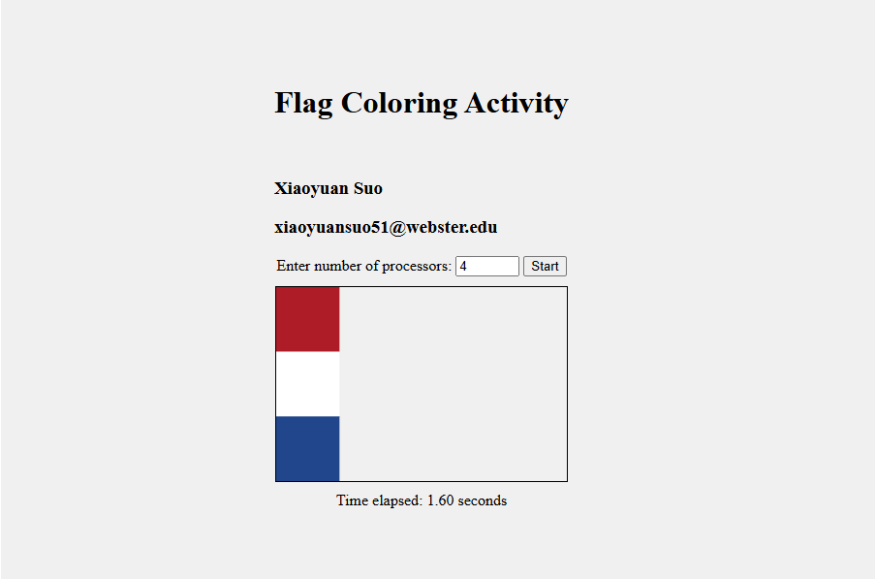


Figure 1.6: Flag Coloring Animation

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.

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.

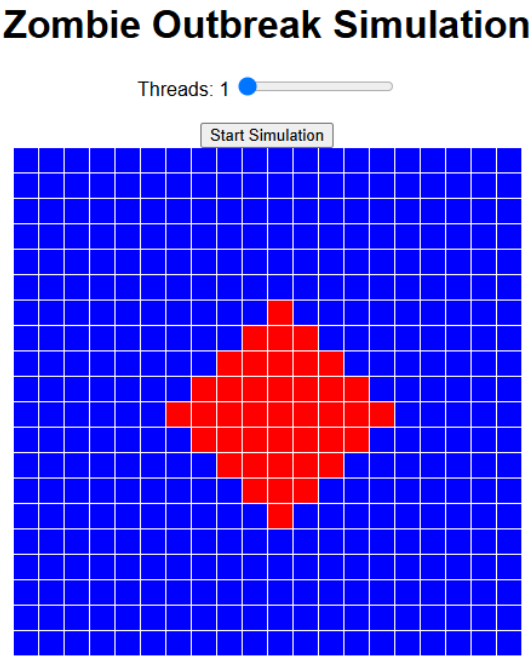


Figure 1.7: Zombie Attack Animation

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 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 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 [50].

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.

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 flag coloring 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 tradeoffs.

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

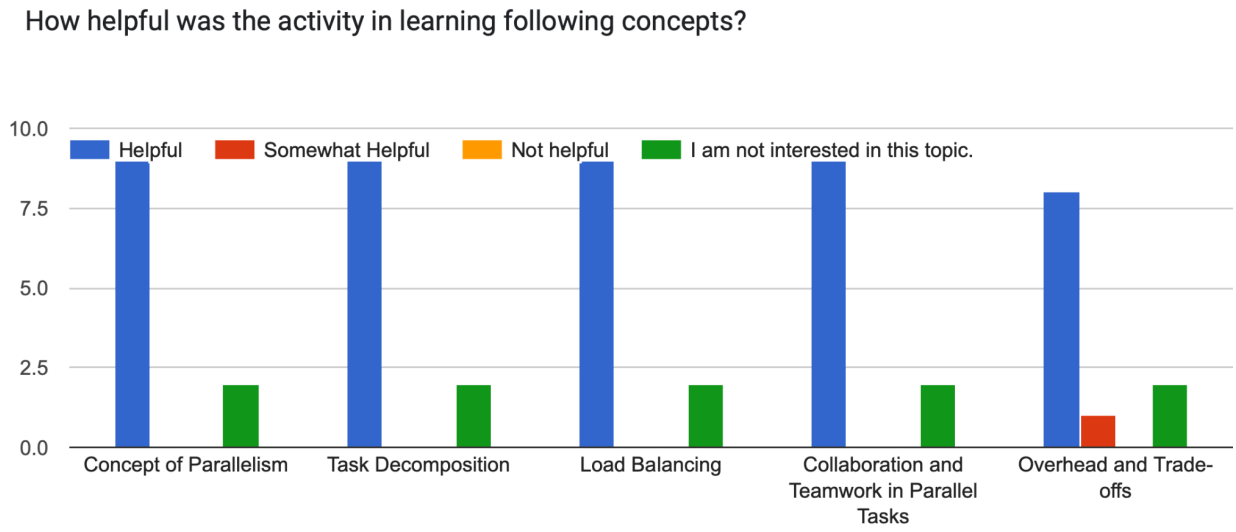


Figure 1.8: Responses from participants

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.

1.3.3.7 Example Implementations

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

- Webster CS1 - Section 6.3.1

1.4 New Plugged-in Activities

1.4.1 Programming Flag Maker

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);
```

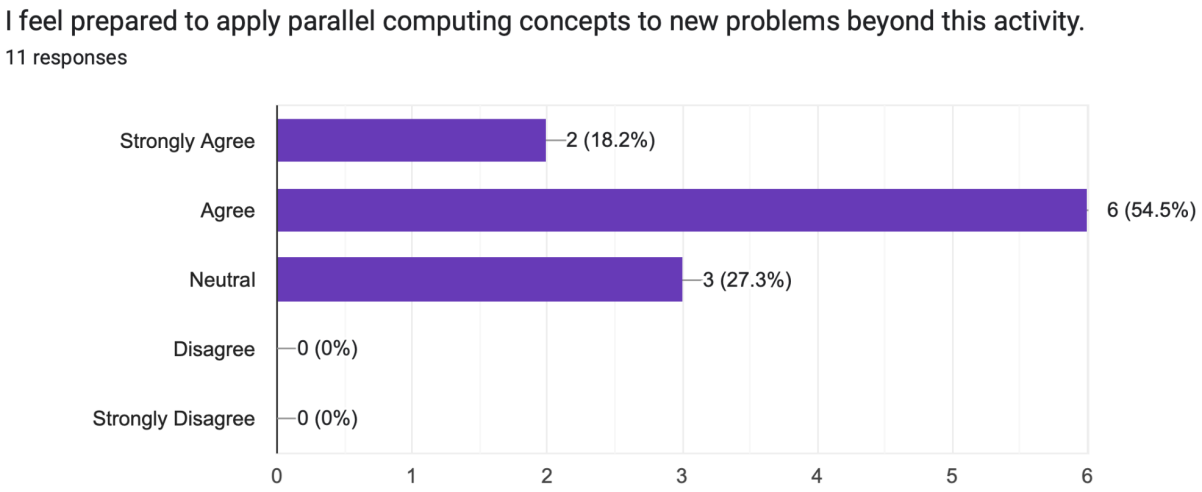


Figure 1.9: The students were then asked if they felt prepared to apply parallel computing concepts to new problems beyond this activity. 18.2 % strongly agreed with it. 54.5% agreed and 27.3 % were neutral.

```
            grid.setColor(r + 2 * R/3, c, Color.RED);
        }
}
```

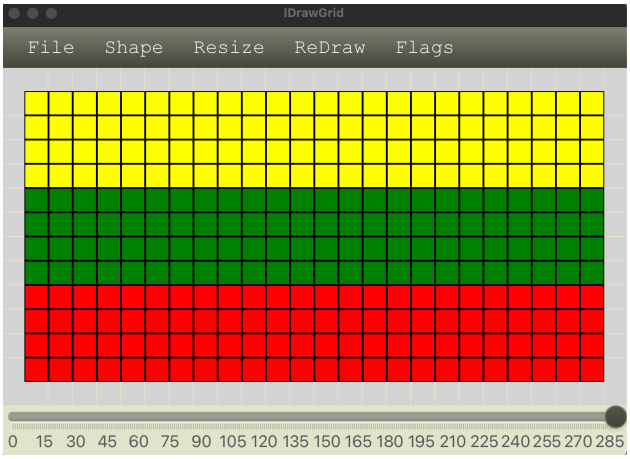


Figure 1.10: The Lithuanian flag, as rendered by the Flag Maker assignment.

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.

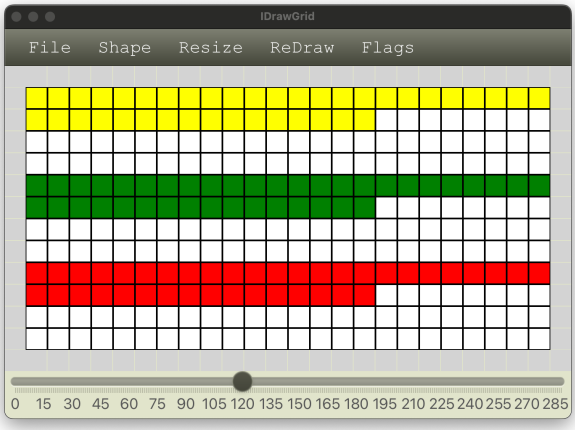


Figure 1.11: The draggable thumb at the bottom replays grid square placement. While not directly parallel, the thumb can nonetheless demonstrate parallelism.

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 [19], 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.

- **How-To Guides**

  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.

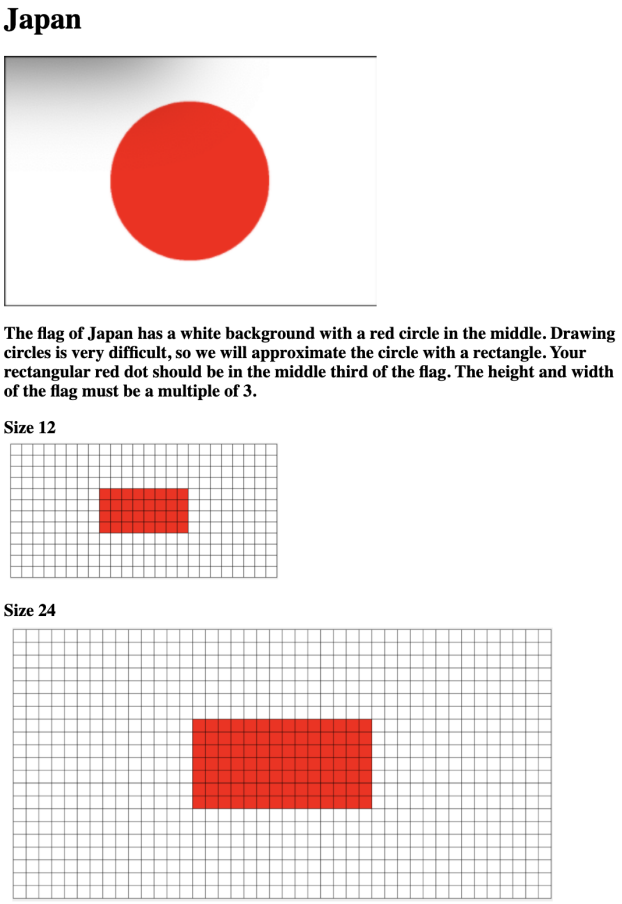


Figure 1.12: The assignment piece for the Japanese flag.

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

```
YYYYYYYYYYY
YYYYYYYYYYY
GGGGGGGGGGG
GGGGGGGGGGG
RRRRRRRRRRR
RRRRRRRRRRR
```

Figure 1.13: Solution as text for the Lithuanian flag.

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 [27, 55] 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.

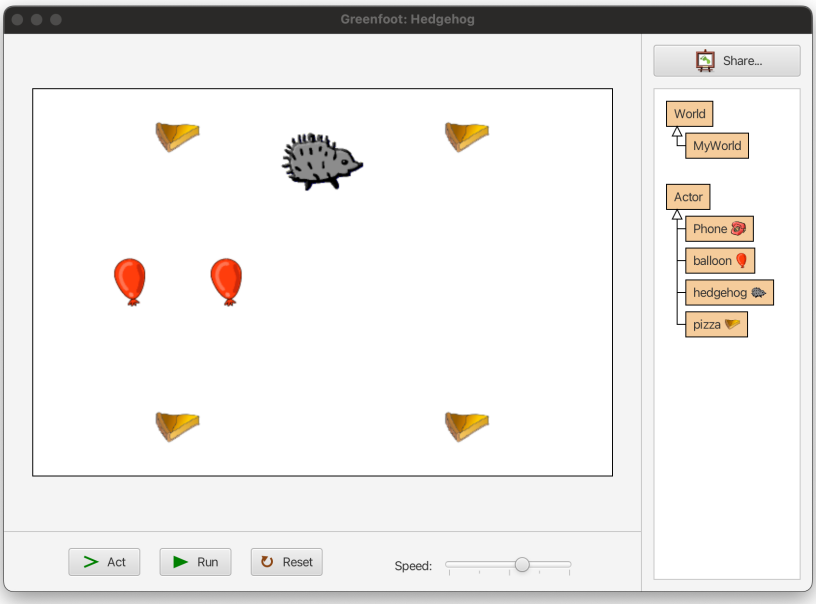


Figure 1.14: The Hedgehog example in Greenfoot. Notice how Actors are clearly organized in the IDE

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 [22] as a Peachy Parallel Assignment [24].

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

```
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.

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.

```
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.

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 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 [25].

Many introductory CS assignments are *well-formed*, *convergent*, and *close-ended* [18, 23]. 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 outweight 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 [40]) 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 is 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, program-

ming 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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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 [54]

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	Earthqu Program 2.4.1	Mod 8 Slides 2.2.2.4	Unplug. Flag Activity 2.3.2	OpenMI 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 [12–15]. 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%)

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 [29, 39]. 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.

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%)

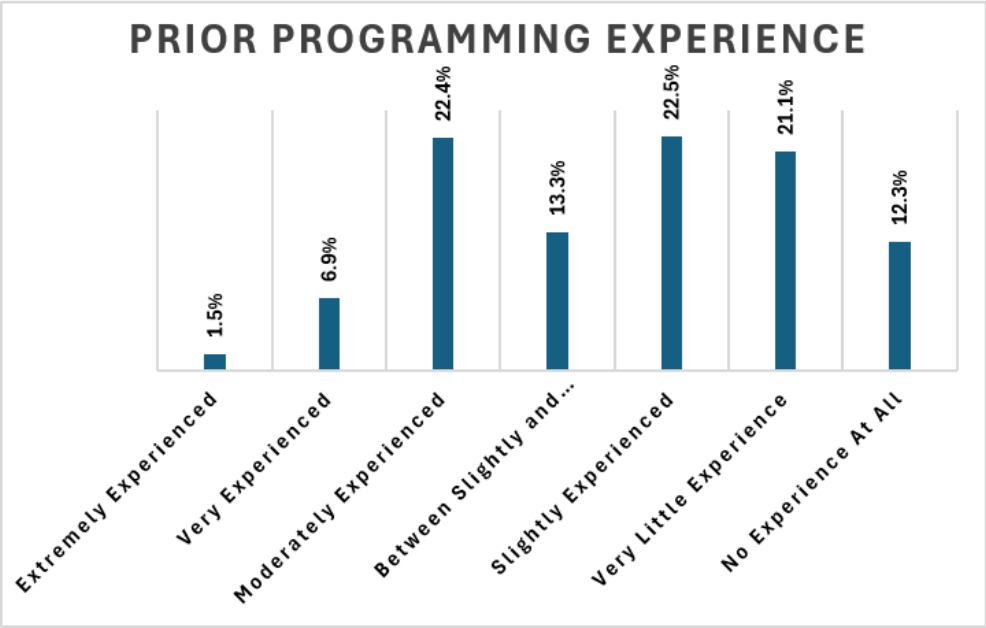


Figure 2.1: Prior Programming Experience

2.2 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 [32],
- cURL library [49],
- Make utility and Makefiles [30]

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.pptx
- 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, 17] 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.

Figure 2.2: Organization of Pennies

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

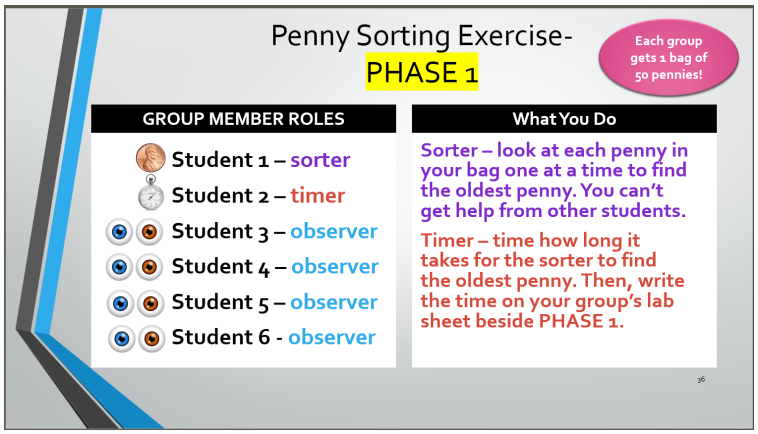


Figure 2.3: Penny Activity Phase 1

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 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 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.

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.

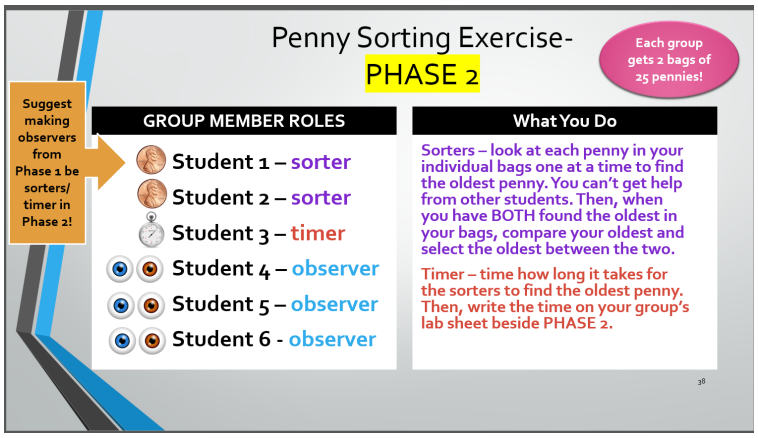


Figure 2.4: Penny Activity Phase 2

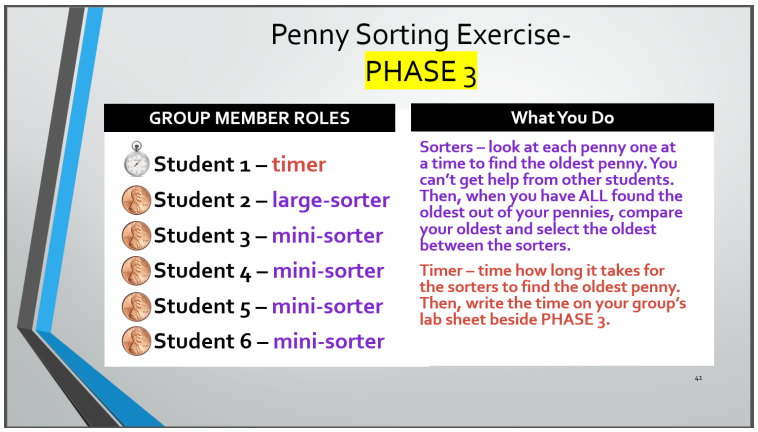


Figure 2.5: Penny Activity Phase 3

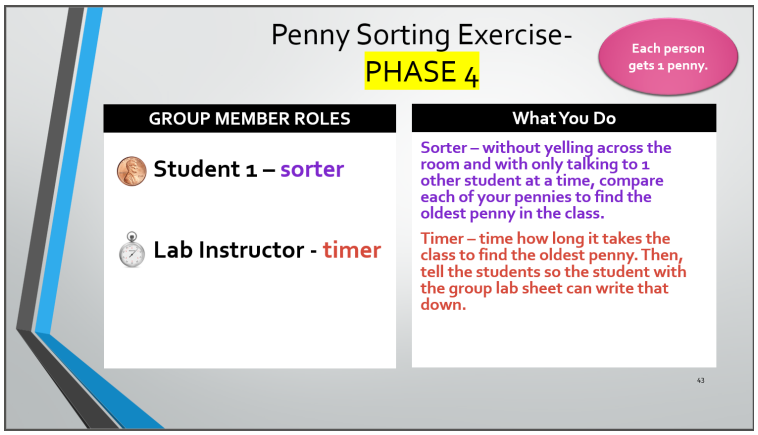


Figure 2.6: Penny Activity Phase 4

2.3.1.1 Instructional Material (Lectures, Assignments)

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

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."

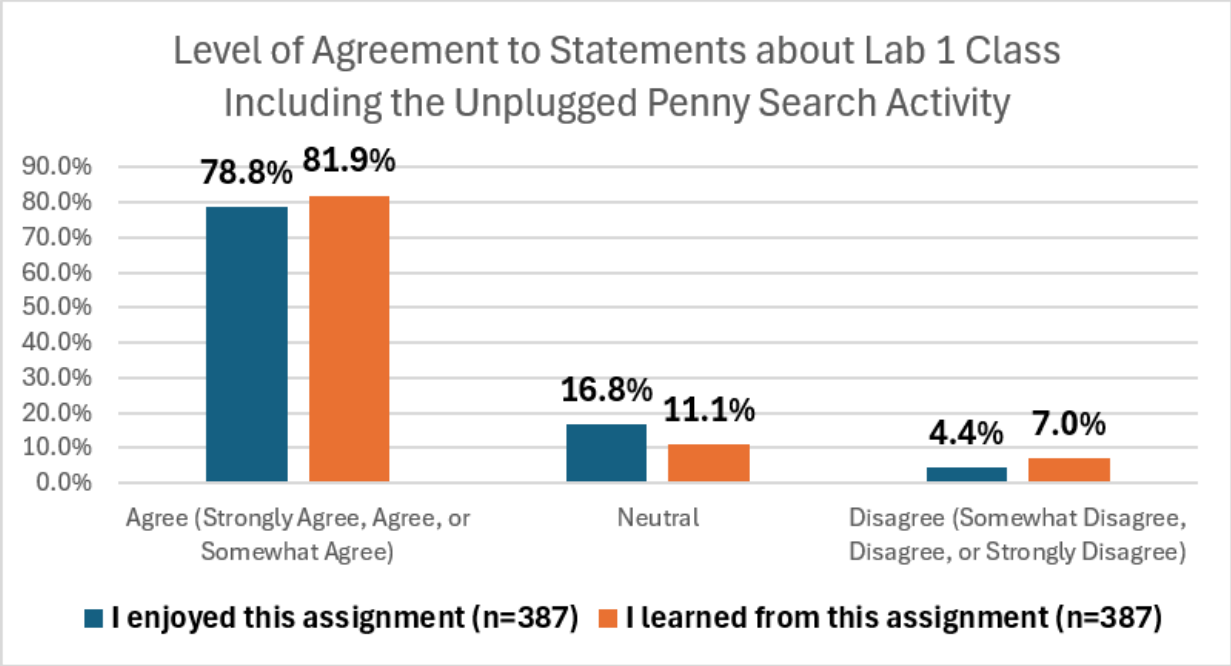


Figure 2.8: Level of Agreement for all of Lab 1 that included Unplugged Penny Activity

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.

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

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

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

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.

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%]

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.

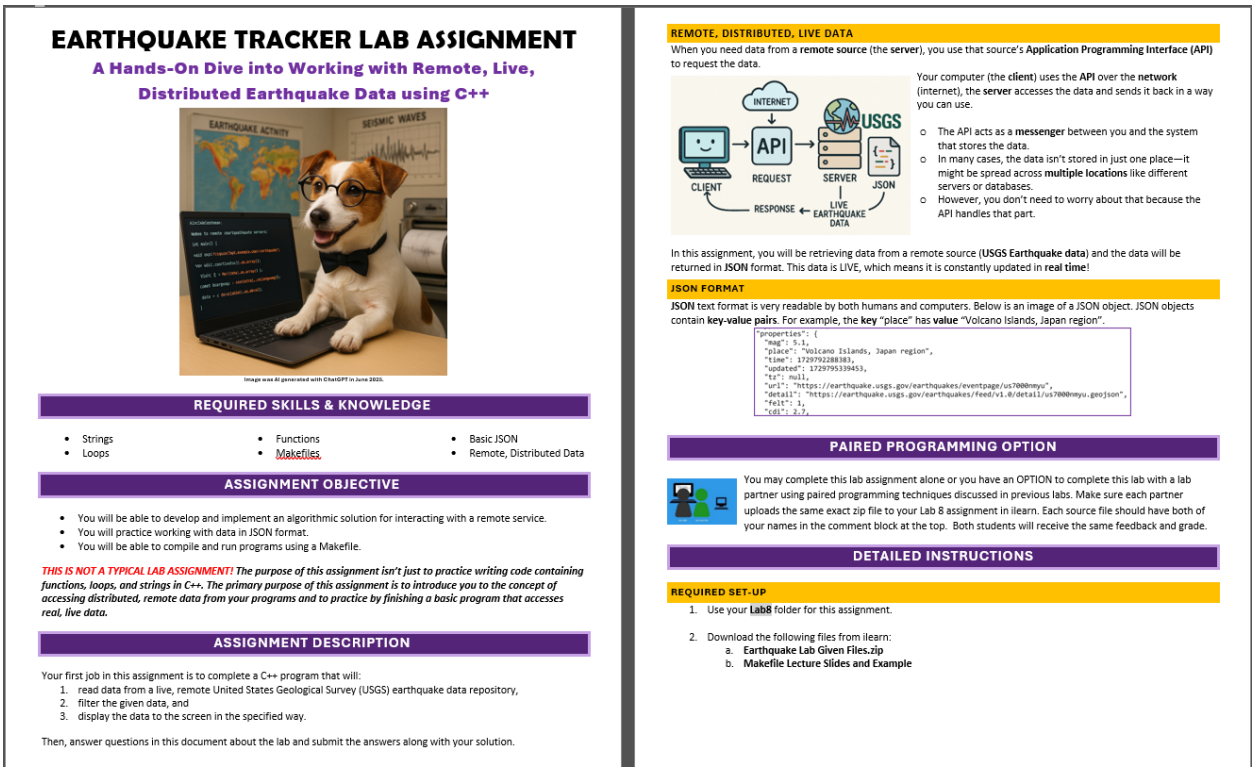
The **learning outcome** for this activity is that students will be able to develop and implement algorithmic solutions for interacting with a remote service.

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 are told not to modify that portion. The inclusion of the exception-handling code allows



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

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.

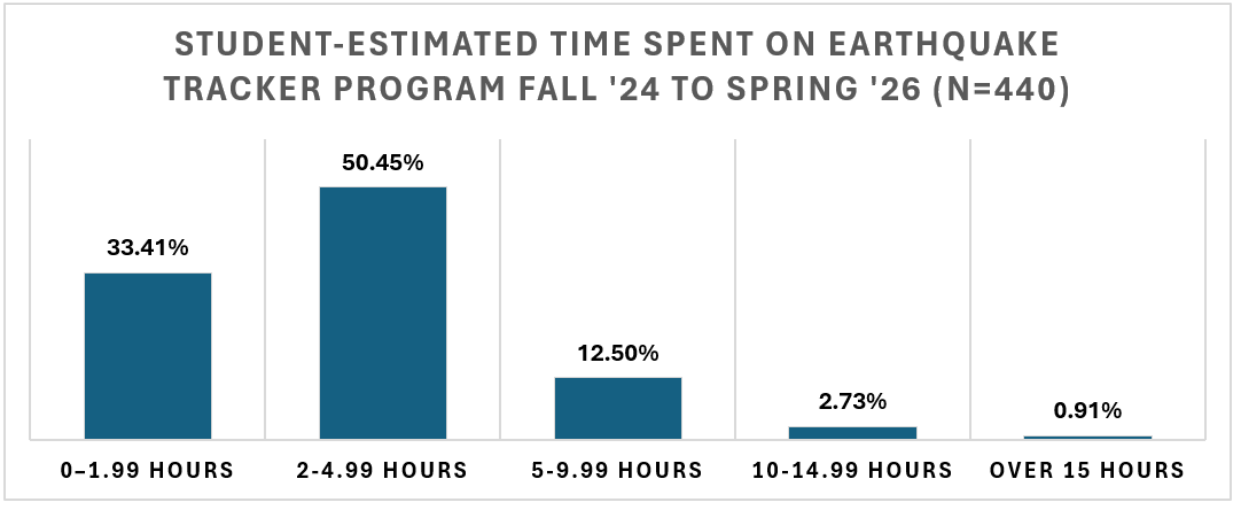


Figure 2.12: Time to Complete Earthquake Assignment (All Semesters)

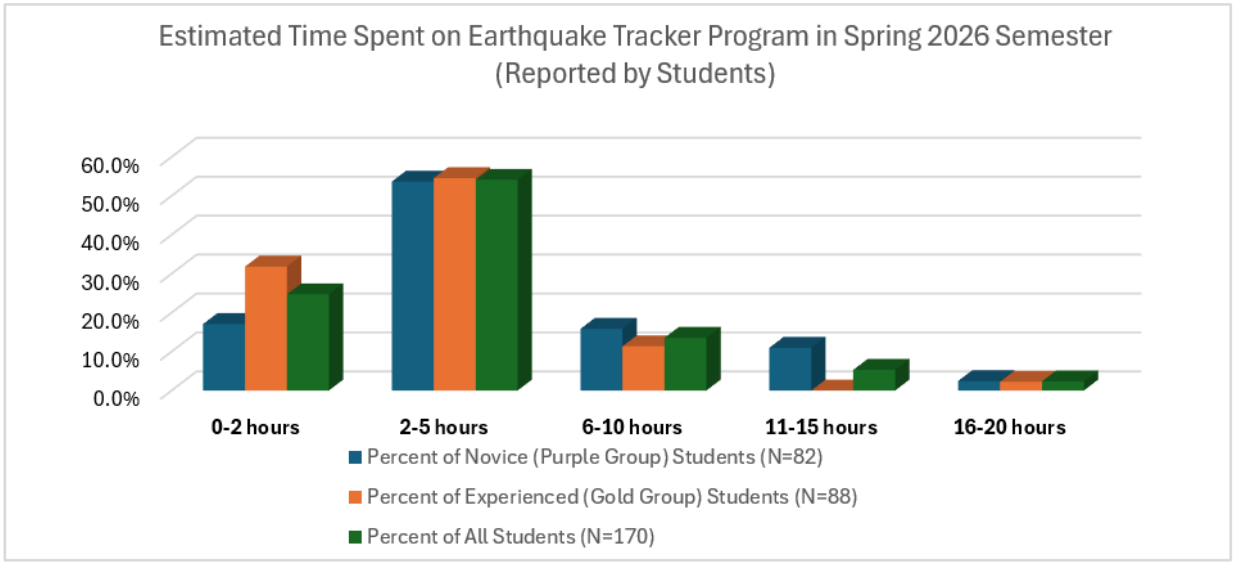


Figure 2.13: Estimated Time to Complete Earthquake Tracker Assignment

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.

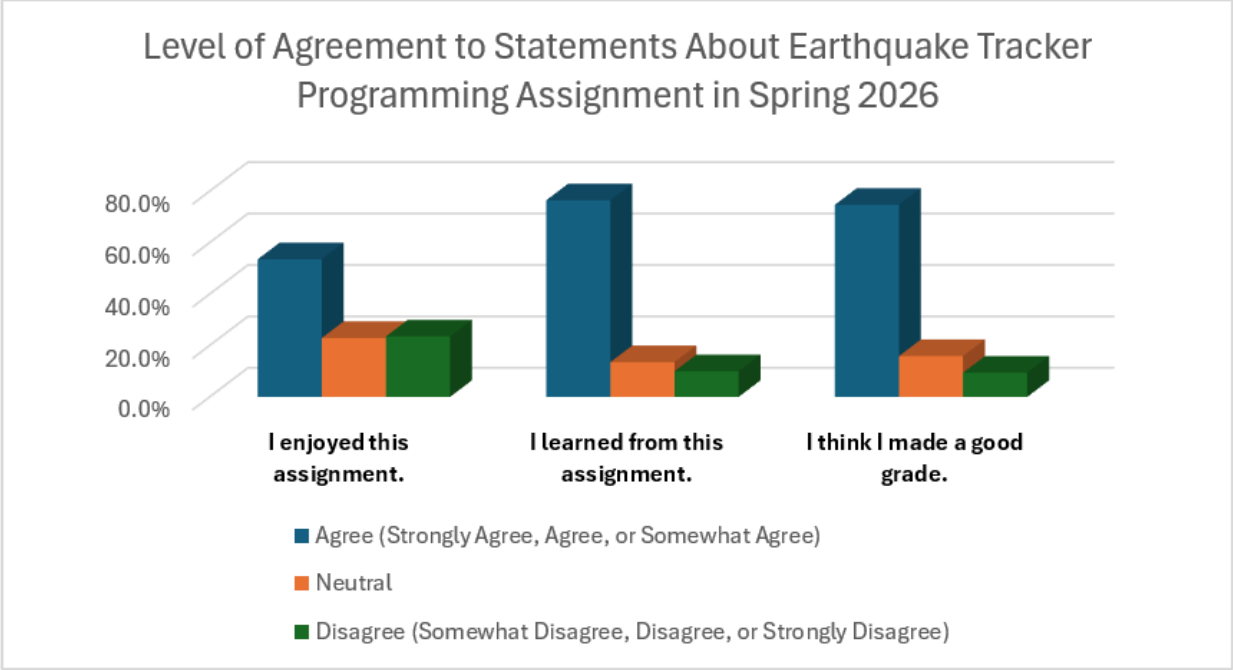


Figure 2.14: Level of Agreement Earthquake Tracker Assignment

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 concepts 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

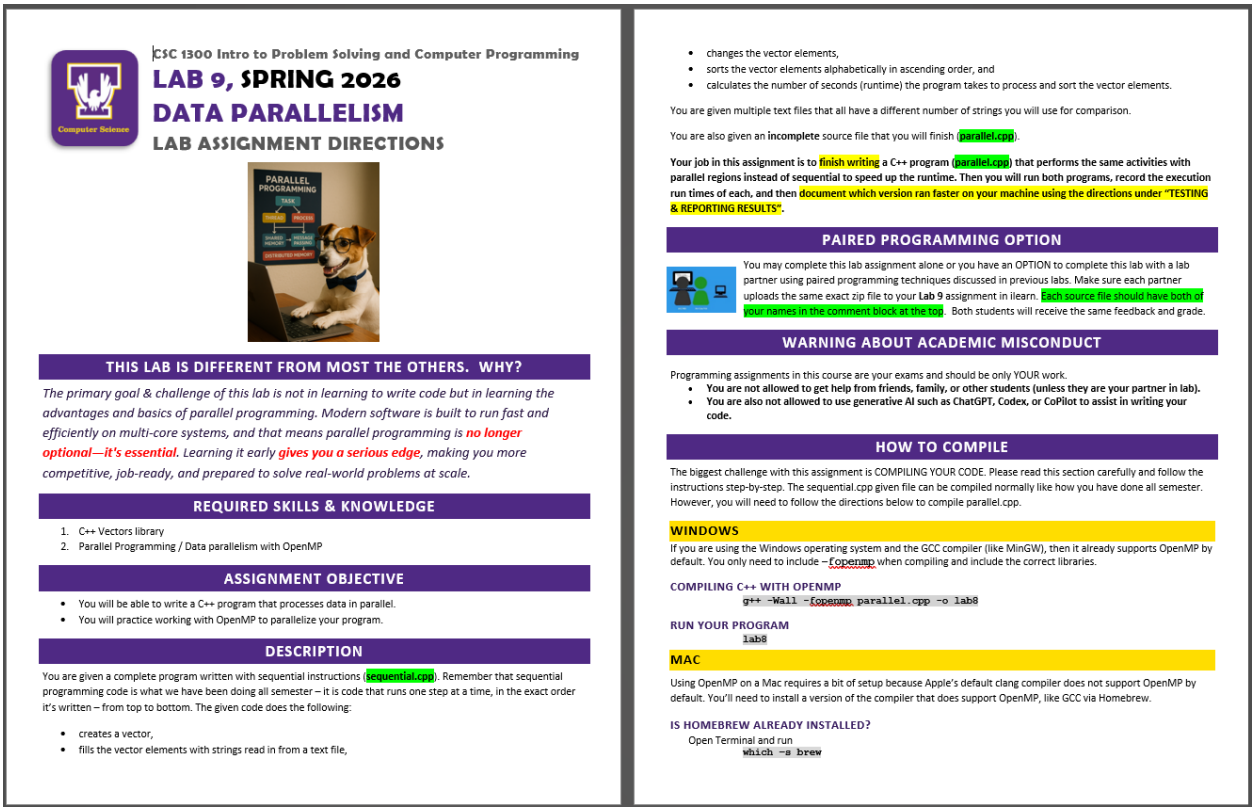
This lab introduces students to data parallelism through a hands-on comparison between a sequential C++ program and an OpenMP-based parallel version. Students are given a completed sequential program, `sequential.cpp`, that reads strings from a text file into a vector, modifies each string by prepending 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.



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

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, 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?"

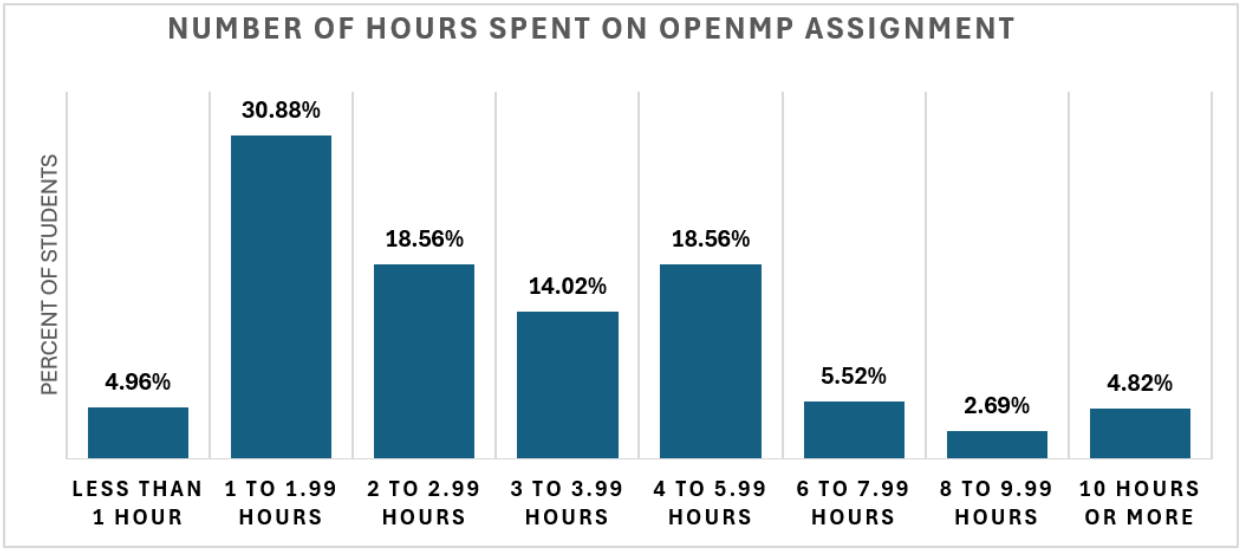


Figure 2.16: Hours Spent on Plugged-In OpenMP Lab Assignment

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.

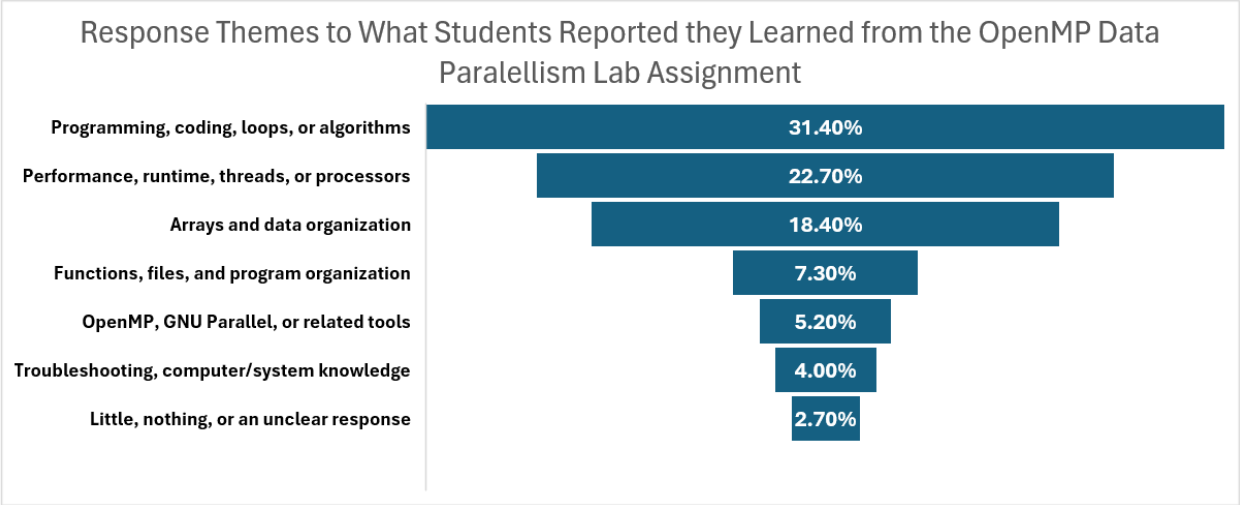


Figure 2.17: What Students Learned from Plugged-In OpenMP Data Parallelism Assignment

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
- 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.

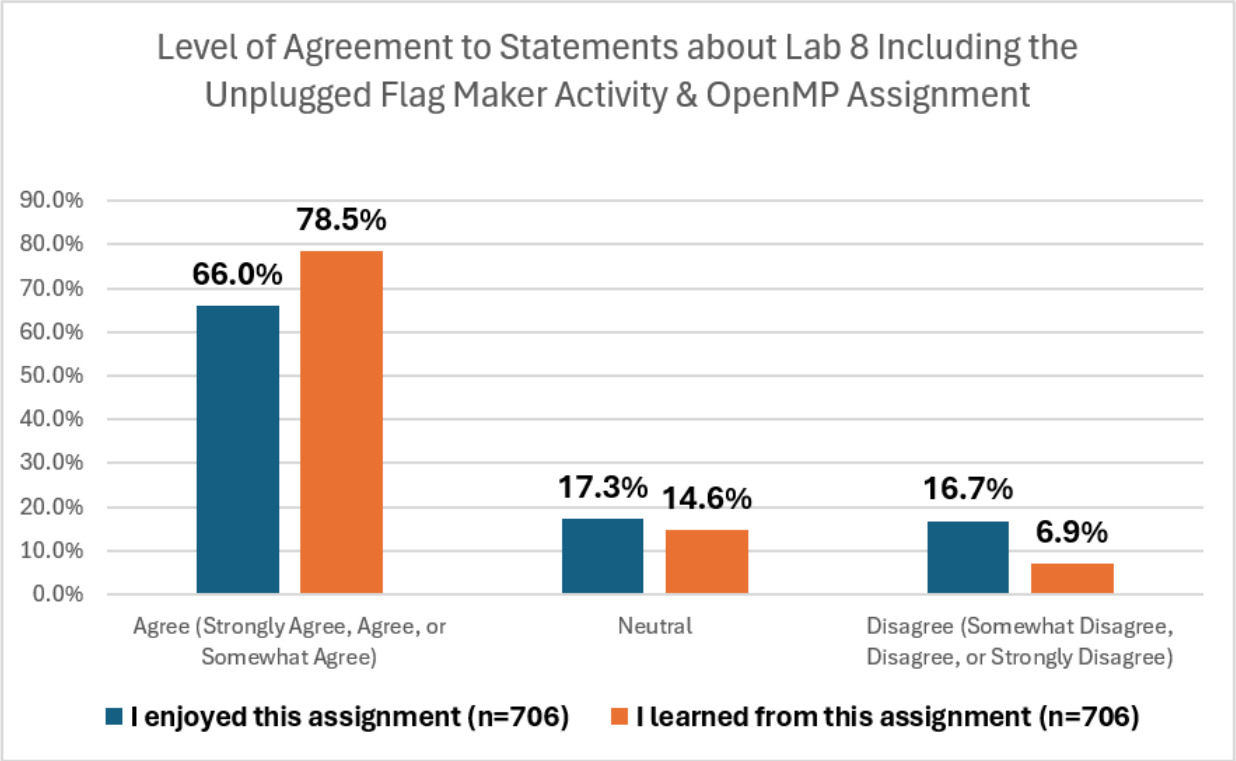


Figure 2.18: Level of Agreement OpenMP Data Parallelism Lab Assignment

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.

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

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%)

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 3.10

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 again demonstrated statistically significant growth across all traditional CS1 concepts, with large effect sizes, indicating that incorporating PDC material did not interfere with their development of 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 intervention, 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 experience 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 2206 CS1 PDC Intervention semesters. `https://github.com/CDER-Center/CS1-CS2_Exemplar_Ebook/tree/main/CS1/chapter2-TTU/Evaluation_Data_and_Analysis/Survey_Instruments`

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

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

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 [36, 37], 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 [11]. 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 [36, 37], 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. [45]; 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

Most of this activity is described elsewhere. 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 told to complete it as homework. The next day, we do the Unplugged Flag Maker Activity (§1.3.1).

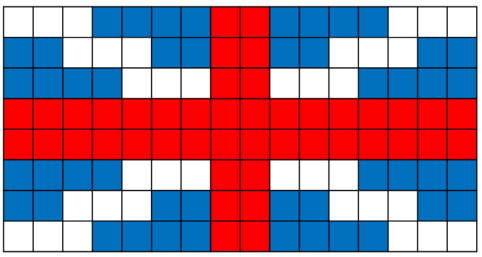


Figure 3.1: Flag coloring assignment version of the flag of Great Britain

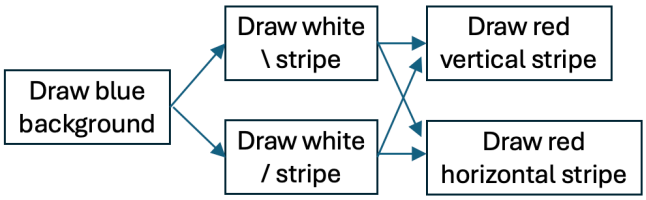


Figure 3.2: Dependency graph for coloring the flag of Great Britain

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 after the unplugged activity.

Another resource that might be useful is a video explaining data parallelism using parallel

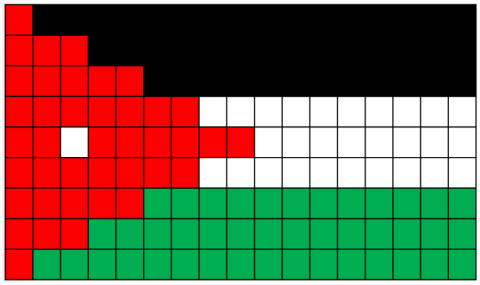


Figure 3.3: Flag coloring assignment version of the flag of Jordan

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 these split the red triangle into upper and lower parts, which was again consistent with how

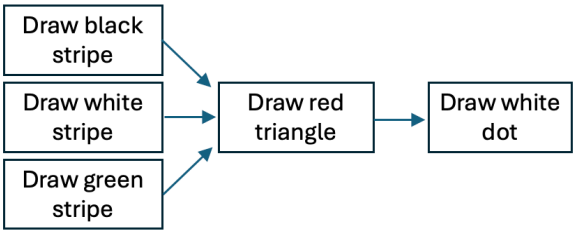


Figure 3.4: Dependency graph for coloring the flag of Jordan

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

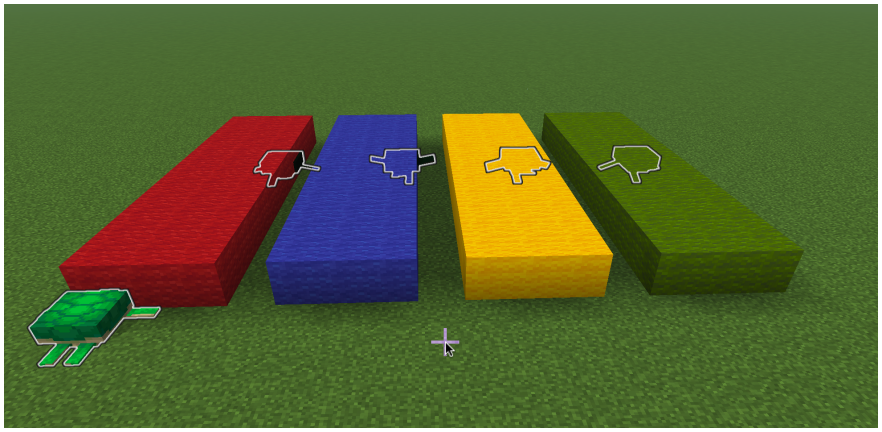
3.3.2.1 Activity Summary

This activity is inspired by Papert's Logo [35], 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

We gave pre- and post-surveys for this course during each term that it was offered, but the results are not included in this early release of the activities. They will be added in a subsequent update.

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 repo.

Detailed Pre and Post Syllabus

Also included are a day-by-day syllabus and discussion of where the new material was added.

Organization of Material

The git repo 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 and a description of the changes made from previous offerings of the course. 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 the activities), `Plugged_Activities/knoxcraft` and `Plugged_Activities/greenfoot`.

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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, 26, 33, 42]. 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 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.

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

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.

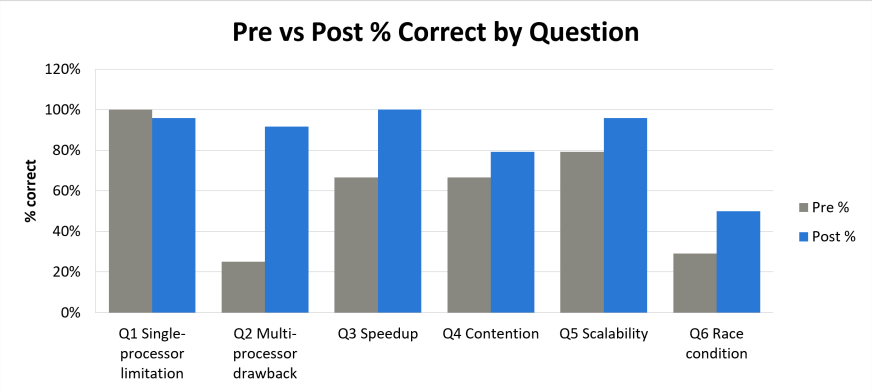


Figure 4.1: USI CS1 Pre vs Post Percentage of Correct Questions for N=24

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.

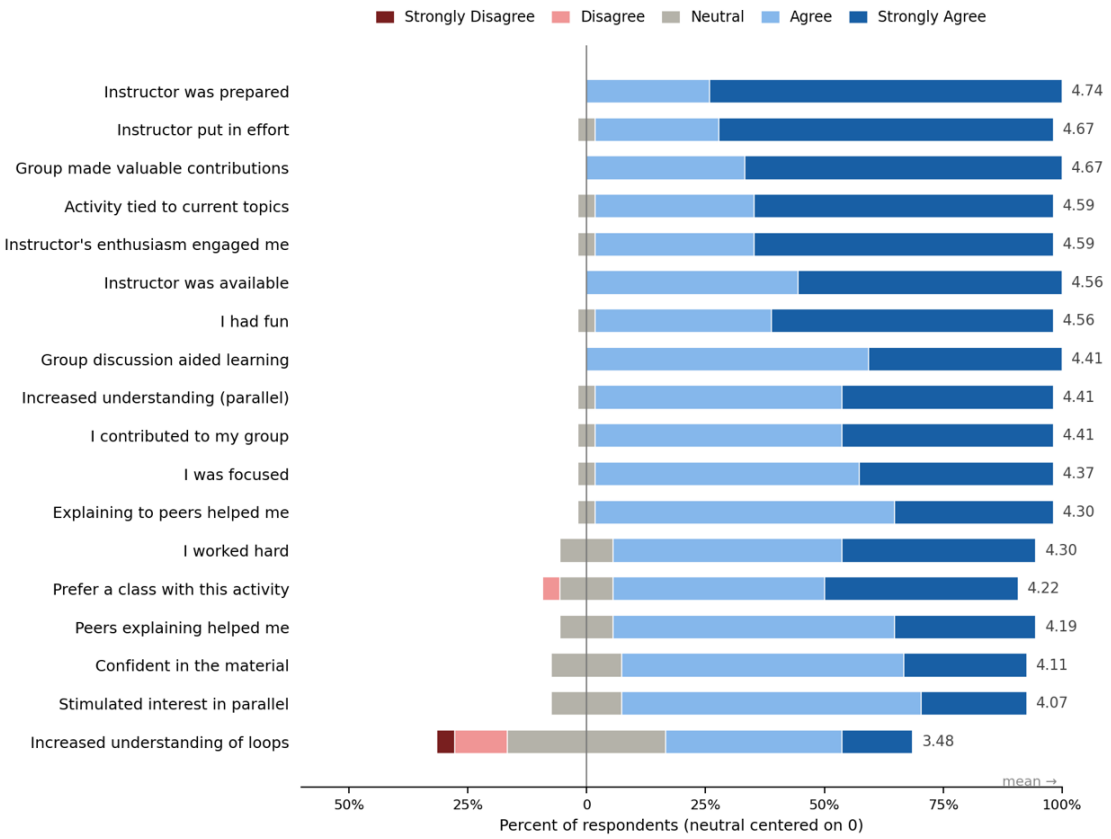


Figure 4.2: USI CS1 Response Distribution by Item

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 a self-contained instructional module structured across four main phases:

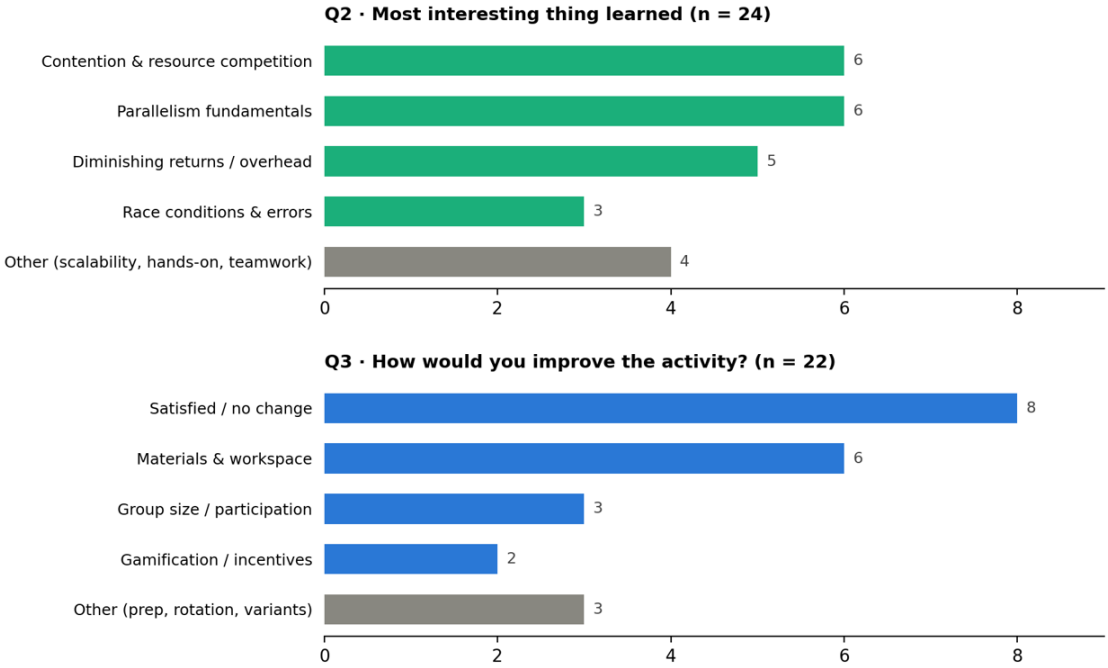


Figure 4.3: USI CS1 Thematic Analysis of Open Response Survey Questions

- 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.

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**

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 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)* [56]. 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.

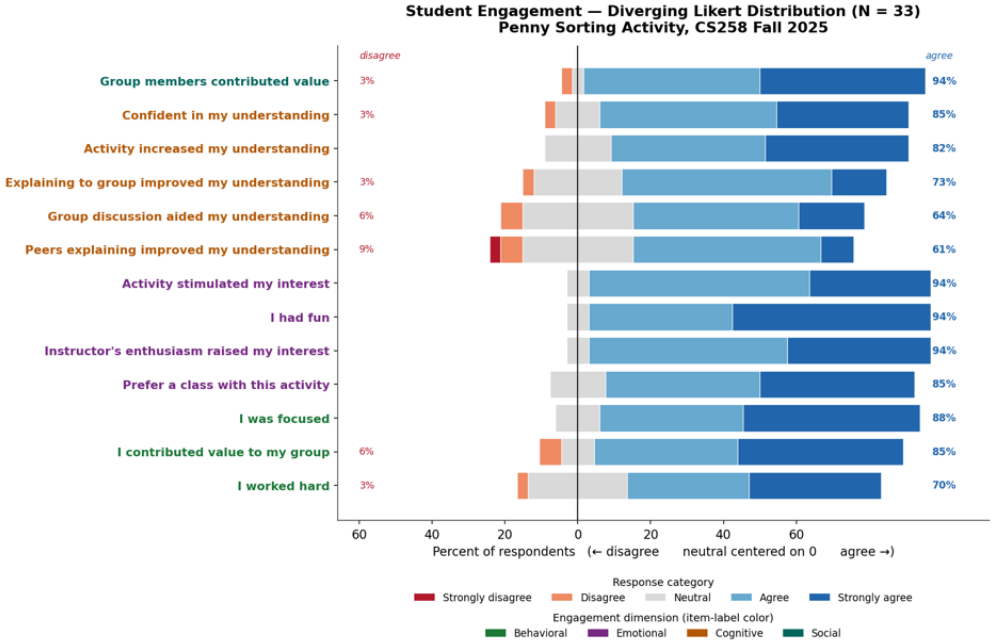


Figure 4.4: USI CS1 Diverging Likert Distribution of Engagement Items (n = 33)

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

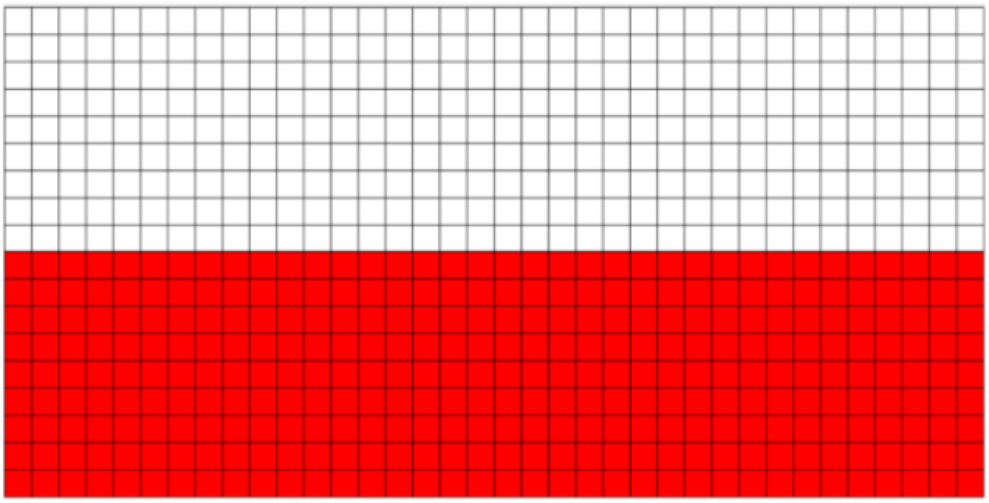
The Flag Maker Plugged-In Activity is a programming assignment in which CS1 students design and implement algorithms to draw a series of international flags using a grid-based graphical environment. Although this activity is described as a PDC intervention for a CS1 course, it is inherently a sequential 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.

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.

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

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.

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

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 decomposition, 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 pre-

sentations, 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 [28], 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

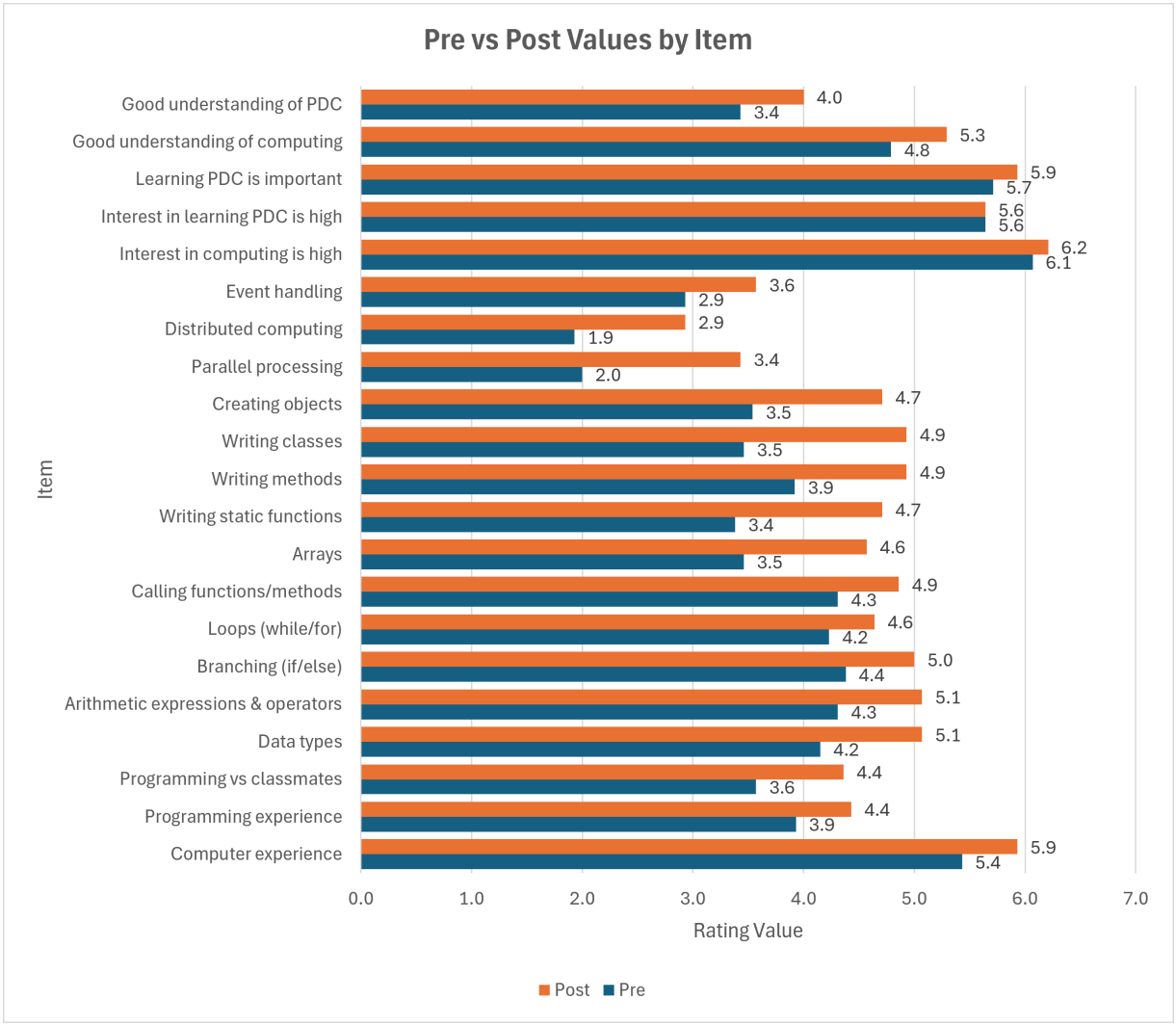


Figure 4.6: USI CS1 Fall 2024 Pre to Post Shift on Every Survey Item

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."

- 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.

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.

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."

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

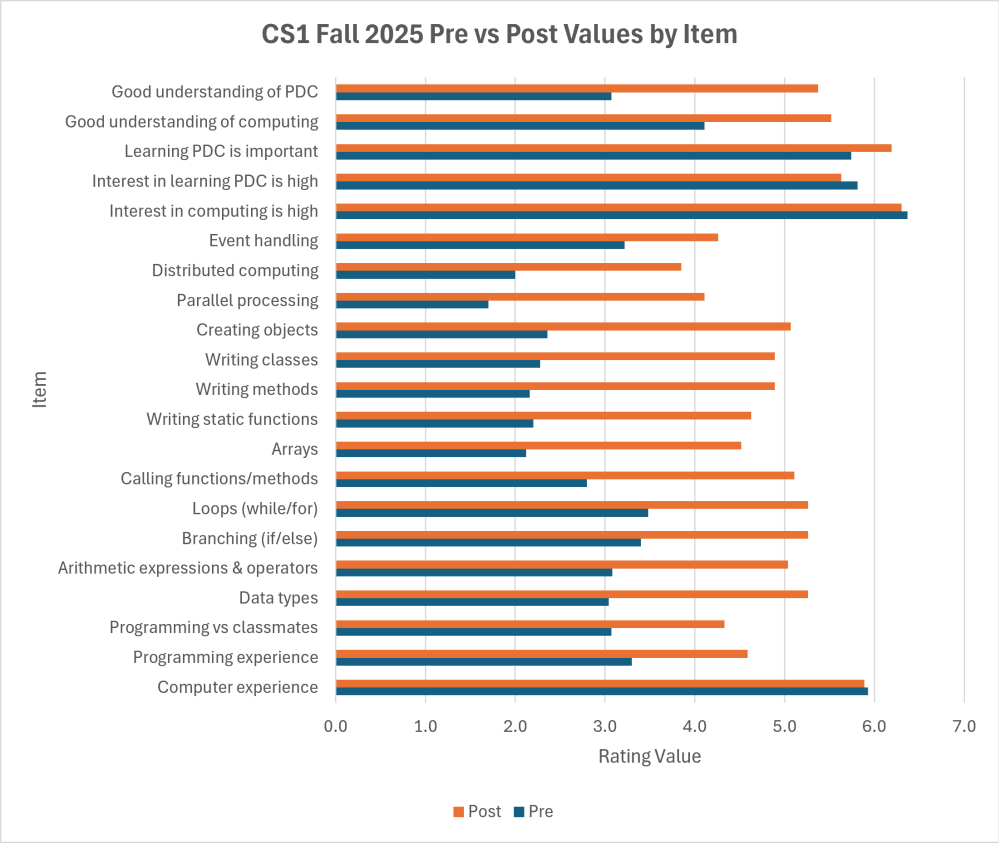


Figure 4.7: USI CS1 Fall 2025 Pre to Post Shift on Every Survey Item

- 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.

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

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

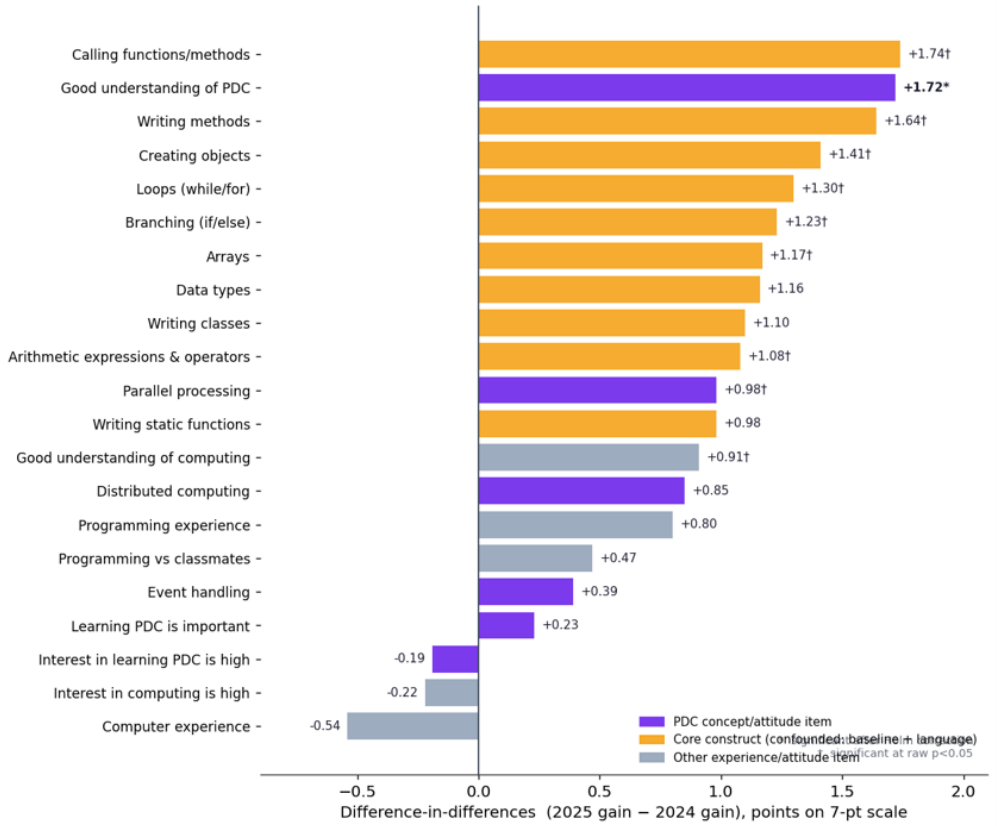


Figure 4.8: USI CS1 Improvement of Each Item from Benchmark Fall 2024 to PDC Intervention Fall 2025

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 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

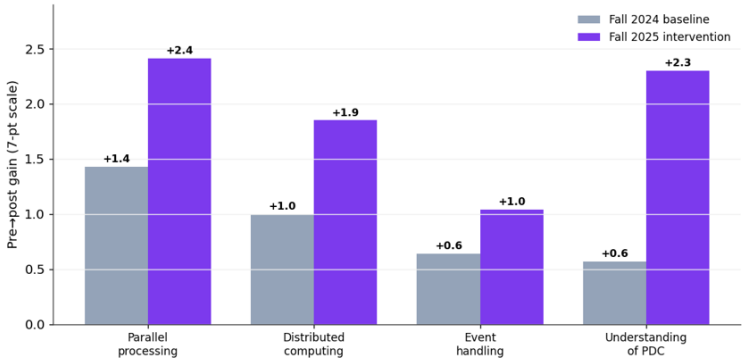


Figure 4.9: USI CS1 Pre to Post Gains on Four PDC Measures: Parallel Processing, Distributed Computing, and PDC Understanding

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.

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

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.

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**

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 [48].

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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:

1. 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 [41, 44, 52]. 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 dfor 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.

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

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 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 [52], data structures and algorithms [44], and automata [41]. 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

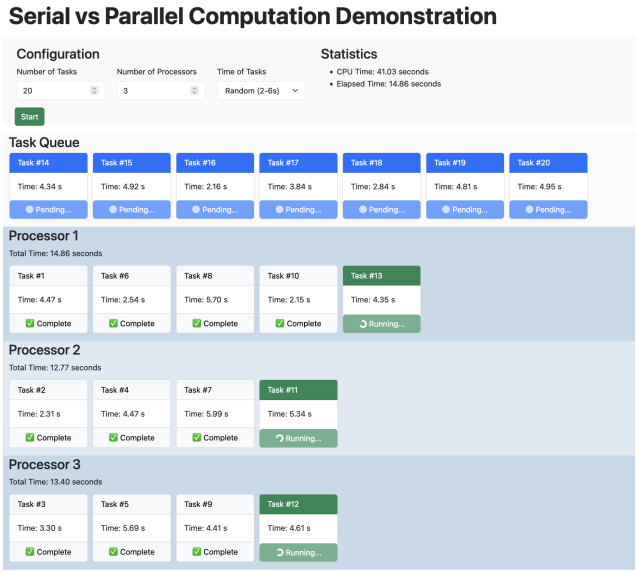


Figure 5.1: A screenshot of Module 1.0's Parallel Computing Demo

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 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,

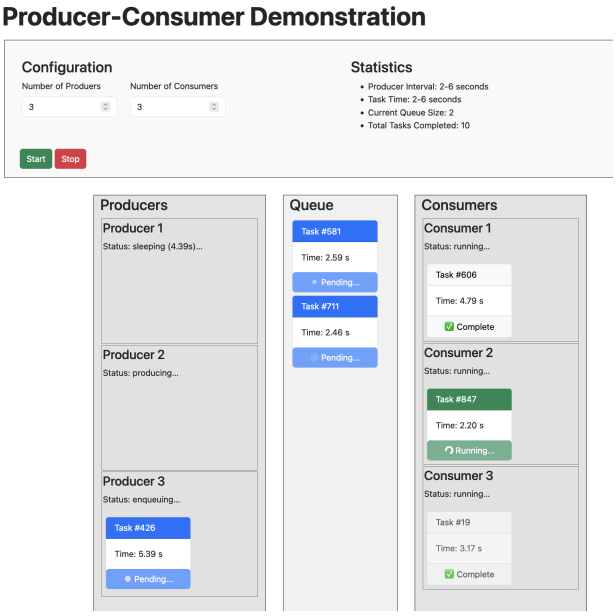


Figure 5.3: A screenshot of Module 3.0's Producer-Consumer Demo

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.

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. However, 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 projects 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.

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. 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 [43].

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

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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

Pre-requisites: no prerequisites

Textbook Starting Out with C++ from Control Structures to Objects, 10th edition, Tony Gaddis [21]

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 [51]. 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 integrates PDC concepts in the later part of the semester (around week 13 of a 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 focus on two carefully chosen examples: an embarrassingly parallel summation problem and a parallel search algorithm. These examples allow students to see immediate connections between sequential and parallel implementations without introducing unnecessary complexity.

6.2 How CS1 was changed and PDC topics integrated

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 Integrated Syllabi (Pre and Post)

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 [34]. 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* 3.3.1 activity[46], 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 a time-consuming task as it usually takes a long time. 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 Demo

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

```
int sum_serial(
    const vector<int>& data)
{
    int sum = 0;

    for (int i = 0;
         i < data.size();
         ++i) {
        sum += data[i];
    }

    return sum;
}
```

Parallel Version

```
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 odd-even parallel Odd-Even sort example

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>  (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
    // Speed up will not show because the data size,
    // should use a much bigger data size
    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 image-processing 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 New Unplugged Activities

6.3.1 Animations and Visualizations

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) and (Parallel Flag Coloring Animation 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.

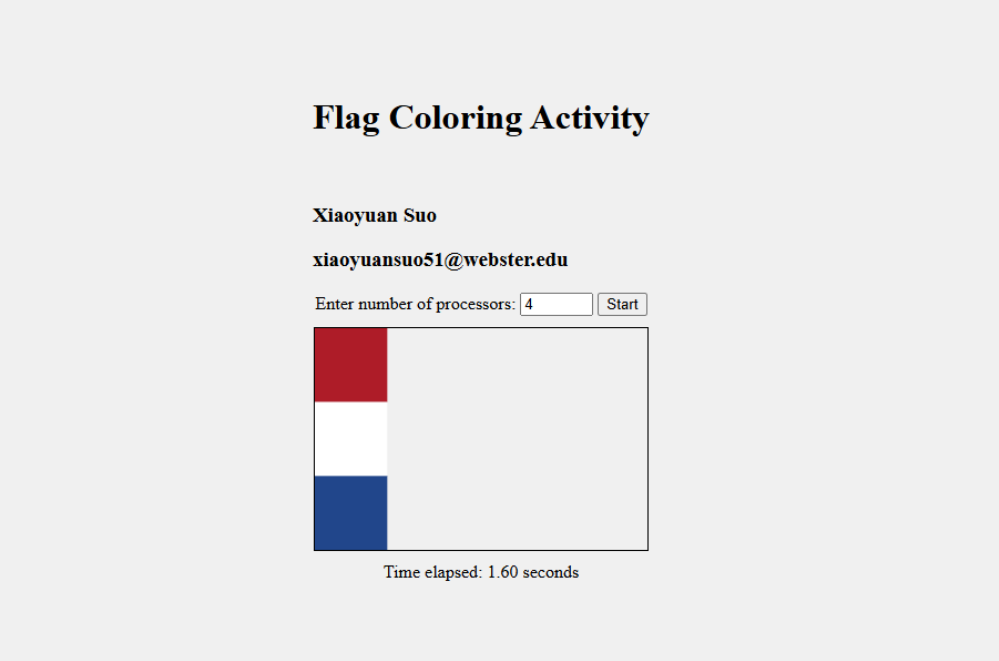


Figure 6.1: Flag coloring animation

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 Plugged-in 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). To clarify, the animation illustrates the spread using a single zombie; however, in the actual assignment, students work

with multiple zombies initialized at random positions on a significantly larger grid. This extension increases the complexity of the simulation and provides a more realistic setting for exploring parallel execution and performance considerations.

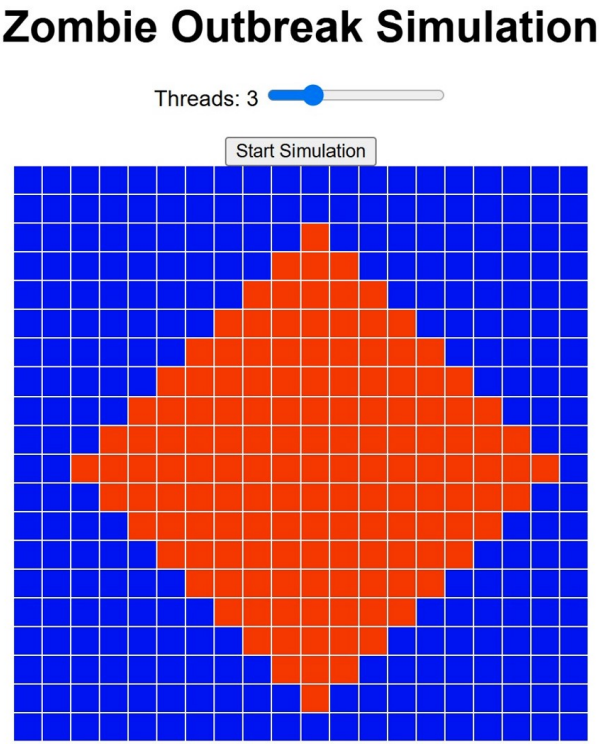


Figure 6.2: Zombie Attack homework: Red grid represents Zombie and Blue grid represents unaffected human

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

A detailed data analysis can be found in the appendix. Please note that the number of students who completed both the pre- and post-surveys was very limited; nevertheless, the survey results indicate measurable gains in both PDC-related familiarity and core programming skills. Across the three PDC-related experience items, the composite mean increased from 1.62 on the pre-survey to 2.21 on the post-survey. Favorable responses (ratings of 5 or higher) increased from 0% to 25% for parallel processing, distributed computing, and event handling. Although students remained novice learners in PDC, the results suggest that the

course successfully introduced concepts that were largely unfamiliar at the beginning of the semester.

Larger gains were observed in foundational C++ programming topics that support later PDC learning. The composite programming-experience mean across ten major C++ topics increased from 4.17 to 5.26. The strongest improvements were observed for structures ($\Delta M = 2.73$), classes and objects ($\Delta M = 1.93$), arithmetic expressions ($\Delta M = 1.09$), pointers ($\Delta M = 0.98$), and arrays ($\Delta M = 0.80$). These gains are particularly relevant because arrays, pointers, and structured data are heavily used in the parallel programming activities and simulations introduced in the course.

Student feedback regarding instructional activities was also highly positive. Programming assignments received the highest post-survey rating ($M = 6.62$), with 100% of students rating them favorably. Attending class ($M = 6.50$) and in-class practice activities ($M = 6.12$) were also rated highly. These findings suggest that the combination of unplugged activities, visual animations, and hands-on programming assignments was effective in supporting student learning and engagement with introductory PDC concepts.

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.

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 of a Textbook Survey responses suggest that students are interested in modern computing topics that extend beyond traditional introductory programming curricula.

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

Topics such as parallelism, simulations, and high-performance computing appeared to increase student engagement because they exposed students to contemporary computing challenges and applications. 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 is missing 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.

6.7 Conclusion

The modules mentioned above required minimal mathematical background and were designed to fit naturally within existing CS1/CS2 curricula without major restructuring. Student feedback and assessment results suggest that approachable, application-driven examples can increase interest and confidence in PDC topics at an early stage of the curriculum. 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 Link to Instructional Material

https://github.com/xiaoyuansuo51-webster/teaching-PDC-HPC

Detailed Pre and Post 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 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. (in class demo)
Final Programming Assignments (Zombie Attack)	Included parallel-thinking exercises emphasizing decomposition, scalability, and workload distribution.

Organization of Material

GitHub Resources and Teaching Materials

The instructional materials, animations, and programming assignments used in this module are publicly available on my GitHub.

- **Parallel Search Animation**

  Parallel Search Animation

This animation demonstrates the concept of parallel search by visually comparing how multiple workers can search simultaneously across a problem space. The activity helps students understand workload decomposition and the motivation for parallel execution.

- **Zombie Attack Simulation Animation**

  Zombie Attack Simulation

  This animation illustrates a simplified grid-based zombie outbreak simulation used to introduce parallel simulations and data decomposition. In the visualization, red cells represent zombies and blue cells represent humans. The animation intentionally uses only a single zombie to simplify the visualization before students transition to the full programming assignment involving large grids and multiple zombies.

- **Flag Coloring Activity**

  Flag Coloring Activity

  This activity uses the Netherlands flag as a simple visual model for discussing parallelism and synchronization. The three horizontal stripes were intentionally chosen to allow students to reason about dividing work into independent regions that can be processed concurrently.

- **Parallel Linked List Activity**

  Parallel Linked List Animation

  This animation demonstrates that although linked lists are inherently sequential data structures and cannot be easily traversed in parallel, different operations or tasks associated with the linked list can still be executed concurrently. The activity helps students distinguish between data-level parallelism and task-level parallelism.

- **Complete Teaching Repository**

  Teaching-PDC-HPC Repository

  This repository contains the instructional materials used throughout the module, including animations, assignments, and supporting input data files. Key materials include:

  - `ZombieAttack_XSUO.html`: Interactive animation introducing the zombie outbreak simulation concept.
  - `Zombie_Assignment_V2.docx`: Parallel programming assignment in C++ using OpenMP.
  - `zombie_input_files.zip`: Sample input datasets for large-scale zombie outbreak simulations.

Table 6.5: Pre/post changes in PDC-related experience. Percent favorable is the percentage of responses coded 5 or higher.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp	d
Parallel processing	1.50	2.25	0.75	0.0	25.0	25.0	0.54
Distributed computing	1.75	2.12	0.38	0.0	25.0	25.0	0.26
Event handling	1.62	2.25	0.62	0.0	25.0	25.0	0.43

Survey and Other Anonymized Data and Analysis

Survey responses were analyzed using a seven-point ordinal scale. For experience items, responses were coded from 1 = no experience to 7 = extremely experienced. For agreement items, responses were coded from 1 = strongly disagree to 7 = strongly agree. Percent favorable was calculated as the percentage of students selecting 5 or higher on the seven-point scale. Because the pre- and post-surveys were anonymous and analyzed as aggregate responses, the results are reported as group-level pre/post changes rather than matched individual gains.

Overall, the data show that students entered the course with relatively strong general programming confidence but limited exposure to parallel and distributed computing (PDC). The composite mean across the three PDC-related experience items increased from 1.62 on the pre-survey to 2.21 on the post-survey, a gain of 0.58 points on the seven-point scale. The percentage of favorable responses increased from 0% to 25% for parallel processing, distributed computing, and event handling. These results suggest that the course introduced students to PDC concepts that were largely unfamiliar at the beginning of the semester.

The strongest quantitative gains occurred in core COSC 1560 programming topics that support later PDC learning. The composite programming-experience mean across ten C++ topics increased from 4.17 to 5.26, for an average gain of 1.08 points. The largest individual gains 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 results are important because arrays, pointers, structures, and object-oriented design provide the technical foundation for later examples involving data decomposition, memory organization, and task-based problem solving.

Student attitudes toward computing remained generally positive. In the post-survey, 87.5% of students rated their interest in computing as favorable, and 87.5% rated their general understanding of computing as favorable. The PDC-specific attitudinal measures were more conservative: 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. This pattern is consistent with a course design in which PDC is introduced as an embedded CS1/CS2-level topic rather than as a full dedicated parallel computing course; students gained initial exposure, but their self-reported mastery remained lower than their general programming confidence.

Perceived helpfulness of course activities. Post-survey responses also indicate that students viewed the main instructional activities as helpful. Programming assignments re-

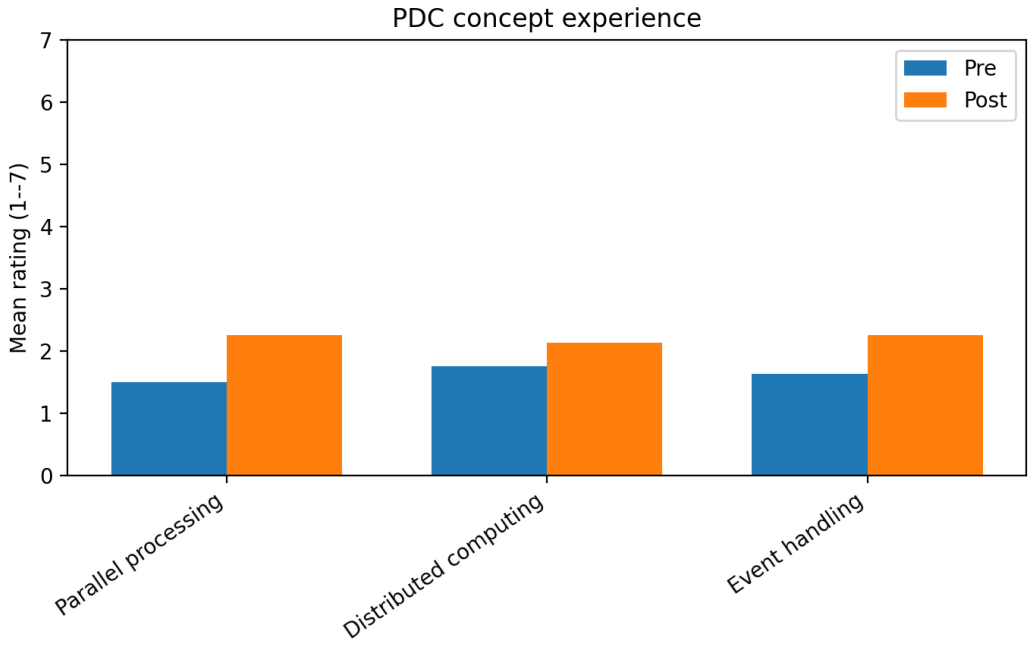


Figure 6.3: Pre/post comparison of student self-reported experience with PDC-related concepts. Mean ratings are reported on a seven-point scale.

ceived the highest rating ($M = 6.62$), with 100% of responses favorable and 100% rated very helpful or extremely helpful. Attending class was similarly strong ($M = 6.50$), followed by in-class practices ($M = 6.12$), textbook use ($M = 5.62$), exam preparation ($M = 5.62$), and meeting with the instructor outside of class ($M = 5.62$). These results support the use of a layered instructional sequence: direct instruction, in-class practice, programming assignments, and assessment preparation.

Post-course learning outcomes. Students reported high confidence on the major COSC 1560 learning outcomes. All students 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 strongest 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 outcomes align well with the PDC module because PDC examples depend on students' ability to reason about data layout, repeated operations over collections, and decomposition of work into smaller units.

The survey results indicate that students showed measurable gains in PDC-related familiarity and larger gains in the programming foundations needed to understand PDC examples. The post-survey activity ratings further suggest that students benefited most from active and practice-oriented components, especially programming assignments, class attendance, and in-class practice.

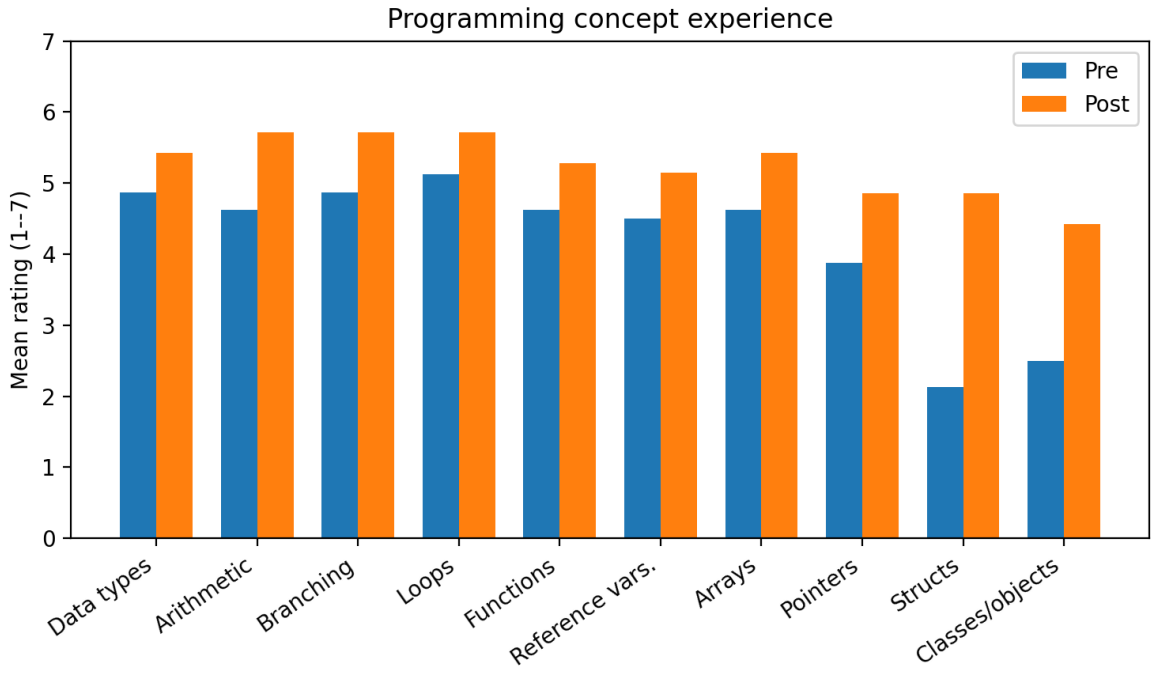


Figure 6.4: Pre/post comparison of student self-reported programming experience across core C++ topics.

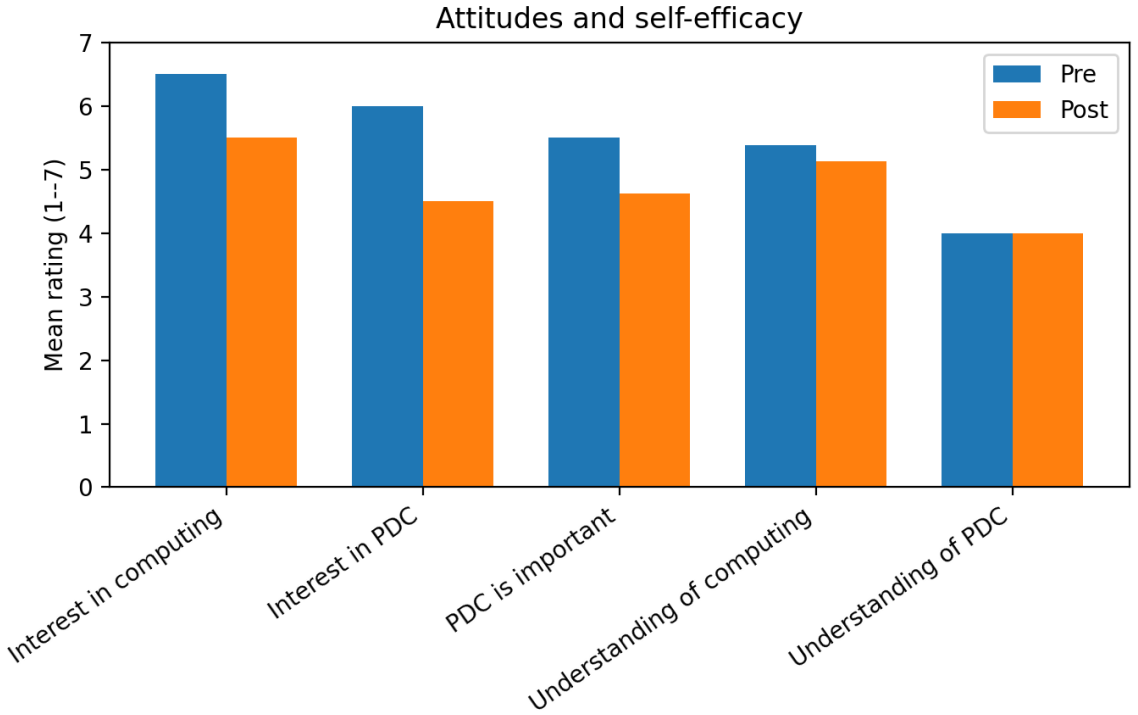


Figure 6.5: Pre/post comparison of student attitudes, interest, and self-reported understanding.

Table 6.6: Pre/post changes in programming experience across core C++ concepts.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp	d
Data types	4.88	5.43	0.55	62.5	85.7	23.2	0.52
Arithmetic	4.62	5.71	1.09	62.5	85.7	23.2	0.81
Branching	4.88	5.71	0.84	75.0	85.7	10.7	0.80
Loops	5.12	5.71	0.59	87.5	100.0	12.5	0.66
Functions	4.62	5.29	0.66	62.5	85.7	23.2	0.78
Reference vars.	4.50	5.14	0.64	62.5	85.7	23.2	0.89
Arrays	4.62	5.43	0.80	75.0	71.4	-3.6	0.85
Pointers	3.88	4.86	0.98	37.5	57.1	19.6	0.76
Structs	2.12	4.86	2.73	12.5	57.1	44.6	2.22
Classes/objects	2.50	4.43	1.93	25.0	42.9	17.9	1.28

Table 6.7: Pre/post changes in interest, perceived importance, and self-reported understanding.

Construct	Pre M	Post M	ΔM	Pre %	Post %	Δ pp	d
Interest in computing	6.50	5.50	-1.00	100.0	87.5	-12.5	-0.88
Interest in PDC	6.00	4.50	-1.50	100.0	50.0	-50.0	-1.20
PDC is important	5.50	4.62	-0.88	75.0	50.0	-25.0	-0.64
Understanding of computing	5.38	5.12	-0.25	87.5	87.5	0.0	-0.26
Understanding of PDC	4.00	4.00	0.00	37.5	37.5	0.0	0.00

Table 6.8: Post-survey ratings of course learning activities.

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 ratings of course learning outcomes.

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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[54] (Summer 2025) and REVEL edition of Starting out with C++ by Tony Gaddis[20] (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.

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

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, 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 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[54] *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[20]. 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++ [54]. 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.

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.

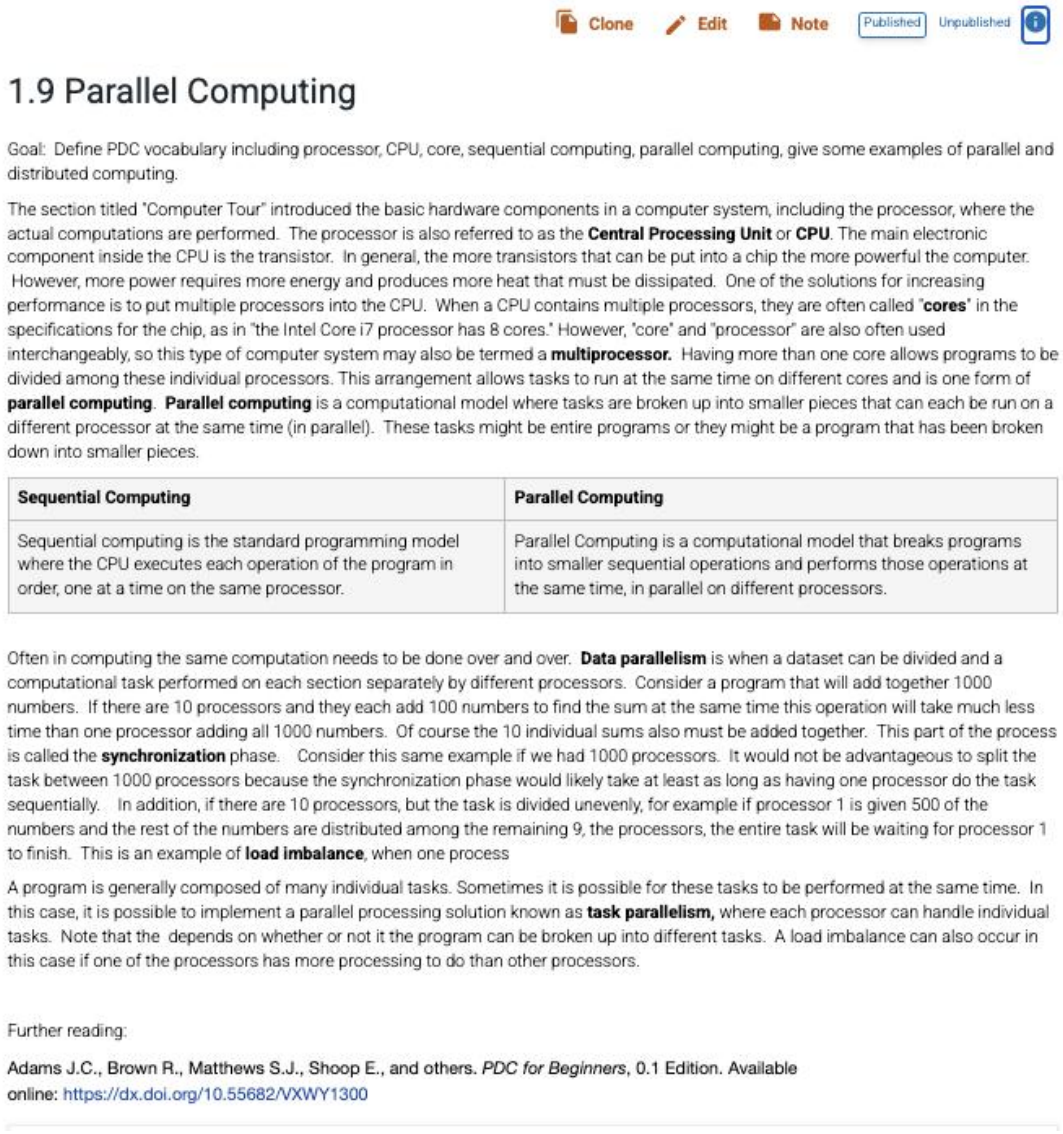
7.3.1.4 Data and Analysis

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



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

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.

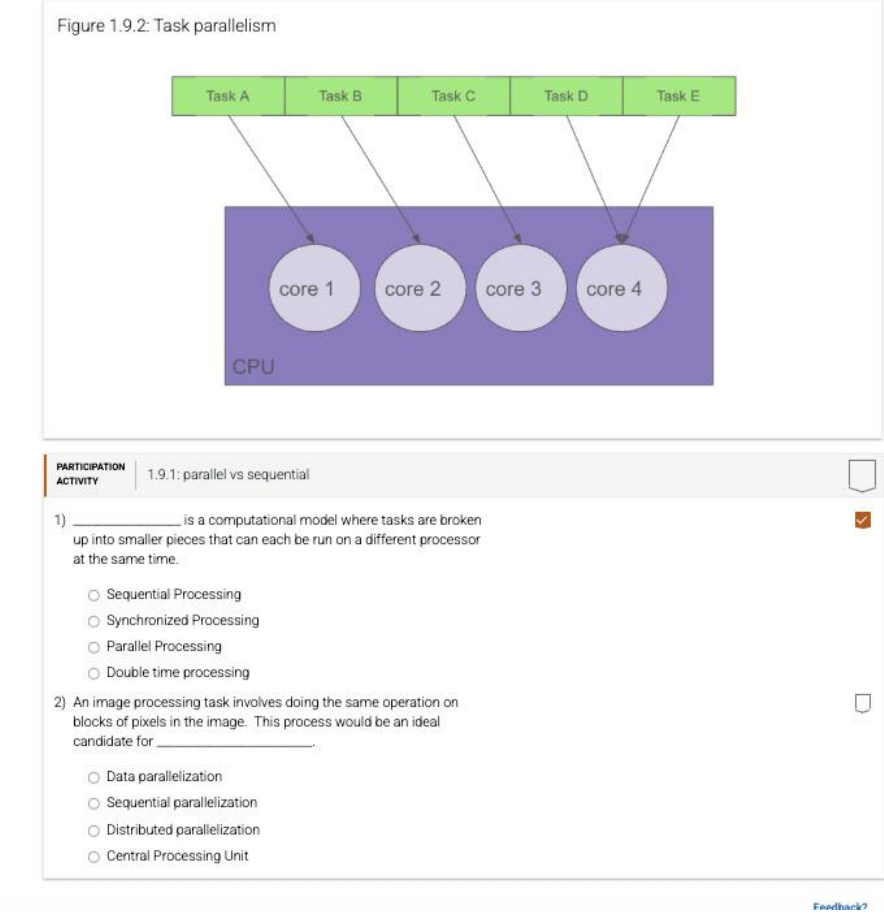


Figure 7.2: Second part of new custom zyBooks section 1.9 Parallel Computing

- 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 acheieved. 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;
    }
}
```

  A video explaining the solution is available here.

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

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, as of the writing of this section, 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. 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

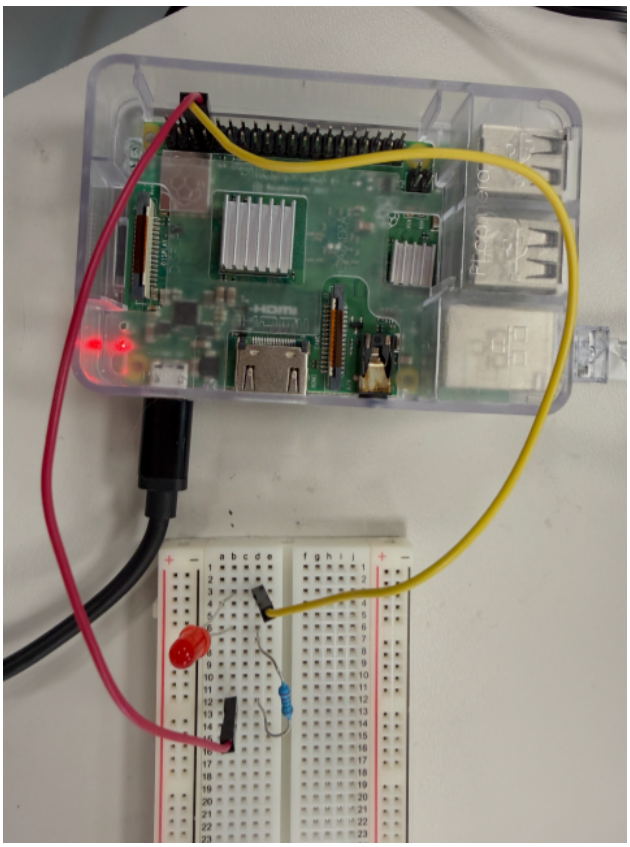


Figure 7.4: CASPER CS1 Hardware Circuit for Physical Computing Project 1

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, 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 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.

```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

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.

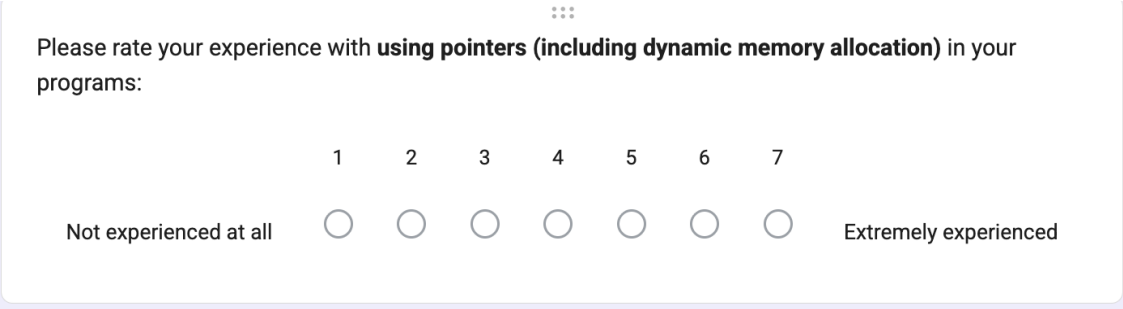


Figure 7.6: CASPER CS1 Sample Survey Question Reflecting on Level of Experience

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

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}	$Mdif_{int}$
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

of various programming topics in their programs. The mean of the difference of the post-pre matched survey responses are in the columns labeled $Mdif\_base$ 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 of the post-pre 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.

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 of the post-pre matched survey responses are in the columns labeled $Mdif\_base$ column. The number of responses for the baseline

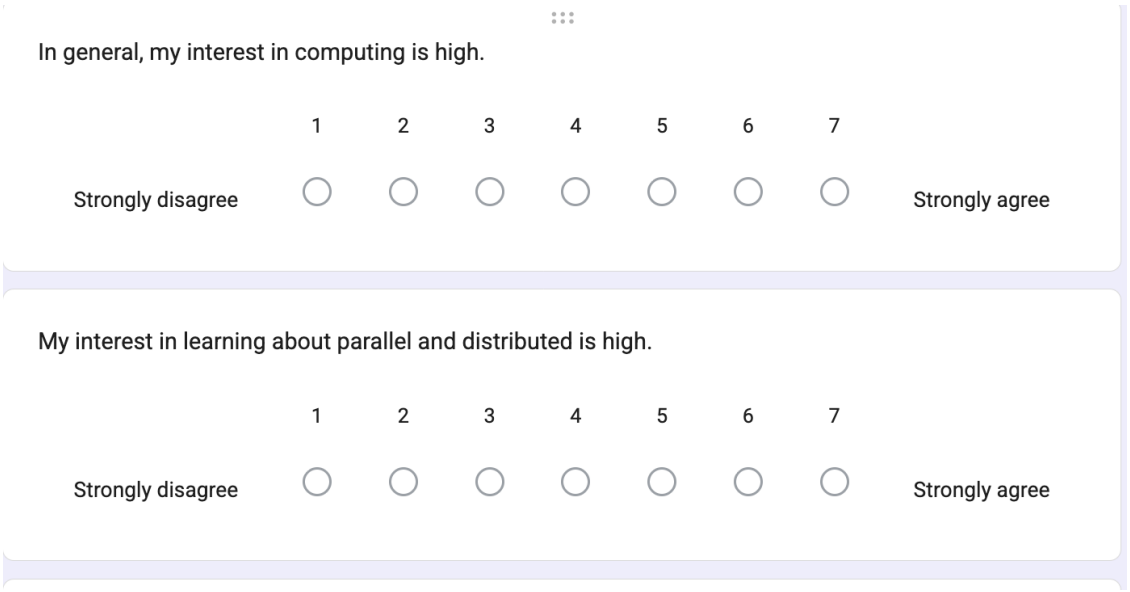


Figure 7.7: CASPER CS1 Sample Survey Question Reflecting on Level of Interest in Computing and PDC

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 of the post-pre 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.

For Table 7.8, the raw responses (on a 7-point Likert scale) for the pre survey R_{base_pre}

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}	$Mdif_{int}$
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

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 of the post-pre 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 unrelia-

> bility 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 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 adverse 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

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter figures

Chapter tables

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	

Continued on next page

Table 8.2 (continued)

Week	Pre-PDC Topics Covered	Post-PDC Topics Covered	PDC Activities
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 instructors 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 facilitator 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 participants (with additional students forming groups of six when necessary). 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 participant colors the entire Mauritius flag sequentially while another participant records the execution time. This phase establishes a sequential performance baseline for comparison with the parallel implementations.

Phase 2 – Two-Processor Parallel Execution: Two participants simultaneously color different portions of the flag by dividing the work by stripes, while a third participant 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 participants each color one stripe of the flag while another participant 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 participants simultaneously color different columns of the flag while another participant 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 facilitator 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) [57]. 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.

The student engagement survey was adapted from the ASPECT [57] 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; however, these demographic responses were not included in the analysis presented in this chapter. The survey questions were grouped according to the pri-

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%

mary 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 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

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

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 ac-

cess 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 work in groups of approximately five or six participants, although smaller groups may also be used with minor modifications. Throughout the activity, participants 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 participant 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 participants, resulting in a load-balanced parallel execution. Both participants search simultaneously, communicate their results, and determine the overall oldest penny while another participant 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 participants, with one participant 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 participant 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 [57] 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; however, these demographic responses were not included in the analysis presented in this chapter. 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 9.2 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 instructor 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 Activ-

ities 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 execution 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 the activity contributed 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 effectively reinforced the concepts introduced through the unplugged exercises by helping students connect the underlying ideas with their implementation in Java.

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. 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.

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

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

topics following instruction. The largest improvement was observed in Writing Static Functions, 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, branching, 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. Furthermore, the relatively small class sizes across the course offerings limit the generalizability of the findings; therefore, the results should be interpreted as descriptive rather than inferential. 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.

The activity is divided into five phases: Phase 1 – Sequential (Uniprocessor) Execution: One participant sequentially writes the sentence *"More Processors Are Not Always The Best"*

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**

and then assigns an index number to each character, while a second participant records the execution time. This phase establishes a baseline for sequential execution.

Phase 2 – Single-Processor Multitasking Execution: One participant alternates between writing each character of the sentence and immediately assigning its corresponding index, while a second participant 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 participant writes the sentence while a second participant simultaneously assigns character indices. A third participant 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 participants divide the writing and indexing tasks into separate portions of the sentence while a fifth participant 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 adjusting 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 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, HPU incorporated several unplugged activities and one plugged activity into an introductory CS1 course to introduce core 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 "parallel 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 parallel and distributed computing. 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 after the integration of the PDC activities

Contents: Post-intervention course syllabus, including a summary of what was changed, course description, learning outcomes, prerequisites, grading policies, and schedules.

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. Aug. 2026 (early release). *Toward Modern Models of Introductory Computing Courses – CS1 and CS2 Course Exemplars infused with Parallel and Distributed Computing Concepts*. CDER Center. NSF Award #2321015. https://doi.org/10.14809/4mcf-5jmt.

Chapter contents

Chapter tables

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, pre-course survey at the beginning of the semester, and post-course survey at the end of the course. Results from paired pre/post survey responses show positive gains in students' self-reported understanding of parallel processing, distributed computing, and PDC-related confidence, suggesting that even a short unplugged intervention can provide meaningful early exposure to PDC concepts in CS1.

Descriptor Elements

Programming Language Employed: Java

Textbook Used: Revel for Java Software Solutions, Foundations of Program Design, 10th Edition by Lewis & Loftus[31]

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 (SLO):

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 multi-processor 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/post activity assessment.

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/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: Conditional 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: Array	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 activity of Flag Maker 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, please refer to Chapter 1, Section 1.3.1 for the Flag Maker activity.

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 pro-

cessing 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.

Table 9.7: Summary of Flag Maker post-activity survey responses.

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%

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 of 6. 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 multi-processor 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.

9.5 Course-wide Pre and Post Course Surveys

To evaluate the effectiveness of 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 FlagMaker intervention semester suggests that the FlagMaker activity did not negatively affect students' learning of 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 writing objects. This suggests that adding the unplugged FlagMaker 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
Writing 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 suggest that students finished the course with slightly stronger self-reported understanding of these PDC-related topics.

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 suggests that the FlagMaker activity helped reinforce PDC awareness while preserving students' 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 non-parametric 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 $\alpha = .05$ level, suggesting that students made measurable progress in these foundational CS1 topics. Several other items did not reach statistical significance, likely due in part to the small matched sample size, but their positive mean differences and medium to large effect sizes still suggest meaningful 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/post 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 suggests that the Flag Maker activity and related course discussion helped increase students' awareness of PDC concepts, even though the short unplugged intervention and small sample size limited the statistical strength of the results.

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 increased, and their self-reported understanding of PDC increased from 2.50 to 3.50, with a medium effect size. The small decrease in self-reported understanding of computing may reflect a recalibration effect, where students became more aware of the complexity of computing after completing more course material.

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 $\alpha = .05$. 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 suggest that students' perceived understanding generally improved 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 show positive gains in students' self-reported PDC understanding, supporting the value of 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 link, lesson plan, code, 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.

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, 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.

[12] 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.

[13] 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. doi: TBD. URL TBD.

[14] 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/.

[15] 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.

[16] 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.

[17] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering, 5(1):46–55, 1998.

[18] Educational Content Team. Convergent thinking. Cogn-IQ Encyclopedia, 2026. URL https://pubscience.org/cqep.2025.0105.

[19] Fawzi Emad. Flag coloring programming assignment. Personal communication to J. Spacco. URL https://www.cs.umd.edu/people/fpe.

[20] Tony Gaddis. Starting Out with C++: From Control Structures through Objects. Pearson, 10th edition, 2021. ISBN 9780135235003. Revel Edition.

[21] Tony Gaddis. Starting Out with C++ from Control Structures to Objects. Pearson, 10th edition, 2023. ISBN 9780137450626. Pearson eText platform.

[22] 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.

[23] Joy Paul Guilford. The structure of intellect. Psychological Bulletin, 53(4):267–293, 1956. doi: 10.1037/h0040755.

[24] NSF/IEEE-TCPP Curriculum Initiative. Peachy parallel assignments. https://tcpp.cs.gsu.edu/curriculum/?q=peachy, viewed July 27, 2026.

[25] David H. Jonassen. Toward a design theory of problem solving. Educational Technology Research and Development, 48(4):63–85, 2000. doi: 10.1007/BF02300500.

[26] 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.

[27] 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.

[28] Michael Kölling. The greenfoot programming environment. ACM Transactions on Computing Education (ToCE), 10(4), 2010.

[29] 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.

[30] Chase Lambert. Makefile Tutorial by Example. Technical report, Makefile Tutorial, July 2026. URL https://makefiletutorial.com/. Online; available at https://makefiletutorial.com/.

[31] John Lewis and William Loftus. Revel for Java Software Solutions: Foundations of Program Design. Pearson, 10th edition, 2025. ISBN 9780138079529. REVEL Access Code Card.

[32] Niels Lohmann. JSON for Modern C++. Technical report, nlohmann/json, July 2025. URL https://json.nlohmann.me/. Online. Available: https://json.nlohmann.me/.

[33] 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.

[34] 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.

[35] Seymour Papert. Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, New York, 1980.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Daniel Stenberg and curl contributors. curl. Technical report, The curl project, July 2026. URL https://curl.se/. Online; available at https://curl.se/.

[50] 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.

[51] 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.

[52] 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.

[53] Wikipedia user Zscout370. Flag of mauritius. https://en.wikipedia.org/wiki/Flag_of_Mauritius#/media/File:Flag_of_Mauritius.svg, viewed Jun 19, 2026.

[54] Frank Vahid and Roman Lysecky. Programming in C++. https://www.zybooks.com/catalog/programming-c-plus-plus/, 09 2025. Interactive online textbook.

[55] 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.

[56] 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.

[57] 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.

PART 2 - Computer Science 2 (CS2)

- ***Expected Soon***