Introducing Parallel Computing in a Second CS Course

Tia Newhall, Kevin C. Webb, Vasanta Chaganti, Andrew Danner
Computer Science Department, Swarthmore College
Swarthmore, PA USA

***Abstract*—The ubiquity of multicore processors, cloud computing, and hardware accelerators have elevated parallel and distributed computing (PDC) topics into fundamental building blocks of the undergraduate CS curriculum. Therefore, it is increasingly important for students to learn a common core of introductory PDC topics and develop parallel thinking skills early in their CS studies. We present the curricular design, pedagogy, and goals of an introductory-level course on computer systems that introduces parallel computing to students who have only a CS1 background. Our course focuses on three curricular goals that serve to integrate the ACM-IEEE TCPP guidelines throughout: a vertical slice through the computer of how it runs a program; evaluating system costs associated with running a program; and taking advantage of the power of parallel computing. We elaborate on the goals and details of our course's key modules, and we discuss our pedagogical approach that includes active-learning techniques. We find that the PDC foundation gained through early exposure in this course helps students gain confidence in their ability to expand and apply their understanding of PDC concepts throughout their CS education.**

***Keywords*-parallel and distributed computing; CS Education; curriculum design.**

I. Introduction

With the increasing ubiquity and application of parallel and distributed computing (PDC), it is imperative to expose students to PDC topics early and often in their CS instruction to provide them with the skills to prepare them for the CS workforce. The ACM-IEEE CS curriculum [1] and NSF/IEEE-TCPP [22] provide guidance on introducing specific PDC knowledge and skills early and throughout the undergraduate curriculum.

As we move from isolated or optional PDC coverage in upper-level electives to more frequent and pervasive coverage, determining how PDC can be incorporated early presents one of the biggest curricular challenges.

We present the design, goals, and outcomes of *Introduction to Computer Systems* [5] (CS 31), an introductory-level course on computer systems that first introduces parallelism. CS 31 is a required introductory course in our curriculum that builds on a CS1 prerequisite. As a required course taught every semester since 2012, CS 31 ensures that all students will have an introduction to parallelism early.

The coverage of CS 31 topics ensures an early exposure to a common core of computer organization, systems, and parallel computing concepts. Students and faculty further leverage this introduction and common core to explore more PDC topics in upper-level courses. An early PDC introduction also helps to meet a broader goal that students are exposed to PDC topics repeatedly, and in different contexts, throughout the curriculum. CS 31 covers many topics from the TCPP curriculum, shown in Table I, spanning Pervasive, Architecture, Programming, and Algorithms areas.

Table I: Main TCCP topics covered in CS 31.

TCPP Category	CS 31 Topics
Pervasive	concurrency, asynchrony, locality, performance in many contexts
Architecture	multicore, caching, latency, bandwidth, atomicity, consistency, coherency, pipeling, instruction execution, memory hierarchy, multithreading, buses, process ID, interrupts
Programming	shared memory parallelization, pthreads, critical sections, producer-consumer, performance improvement, synchronization, deadlock, race conditions, memory data layout, spatial and temporal locality, signals
Algorithms	dependencies, space/memory, speedup, Amdahl's Law, synchronization, efficiency

II. Curriculum Overview and Goals

Our introductory sequence consists of three courses: our CS1 course in Python; CS 31, an introduction to computer systems; and CS 35, data structures and algorithms. Students can take CS 31 and CS 35 in any order or concurrently; CS1 is the only prerequisite for both.

CS 31 is a broad introduction to computer systems and to parallelization focusing on parallelism for multicore systems. Its curriculum is designed around three main themes.

The first theme is understanding **how a computer runs a program**. This content describes a vertical slice through the computer that examines how the high-level language C is compiled to binary instructions that are executed on CPU circuitry. The course also examines the role of the Operating System (OS) and its abstractions in running programs.

The second theme **evaluates systems costs associated with running a program.** We focus primarily on the performance effects of the memory hierarchy, but we also discuss other system costs including the OS's role in scheduling for efficiency as well as synchronization and parallelization overheads.

The third theme is **taking advantage of the power of parallel computing.** We focus this introduction on shared

memory parallelism. This fits well with our model of "the computer" that we use throughout the course, where students become familiar with the details of both single core and multicore computers. We stress "thinking in parallel" as we cover race conditions, synchronization, deadlock, speed-up, the producer-consumer problem, and designing and implementing parallel programs in pthreads.

Beyond our primary themes, students also become proficient in implementing solutions to large problems in C and in using GDB and Valgrind to debug and examine program execution state.

Course Structure. The structure of CS 31 includes lectures, larger programming lab assignments, written homeworks, in-class group exercises, and two course exams. CS 31 also includes a weekly 90 minute lab section covering programming tools and providing hands on experience applying lecture topics.

We use the free, online "Dive into Systems" [15] textbook, written by two of the co-authors and a collaborator from West Point. Motivated by a lack of options, the book covers *breadth* of topics in CS 31 at an appropriate *introductory depth*. In addition to its use in CS 31, the book is designed to be a useful resource to a broad range of courses that introduce computer systems, computer organization, and parallel computing [16].

CS 31 class meetings consist of lectures punctuated by several active learning exercises in small groups. We adopt the peer instruction [19] teaching model and use student clicker devices to poll the class. We present a carefully crafted question and first ask the students to answer it individually. For full participation credit, the students must commit to an answer via the clicker, but we don't grade it for correctness. Next, we give students 2–3 minutes to discuss the question in small groups and then respond again via their clickers, this time answering as a group. Finally, the instructor holds a full-class discussion, calling on volunteers to share what their group discussed.

Prior to class, we ask that students read brief introductory material from a textbook, and we hold daily (graded) reading quizzes that students answer via their clicker. These quizzes are designed to be answerable by students who did the reading, even if they don't yet hold a deep understanding of the content.

During our weekly lab sessions, we begin by working through warm-up exercises. As the first systems course that students take, our aim with the in-lab instruction and warm-up exercises is to provide students with a way to experiment, learn, and build confidence in C programming and system top-down design without having the pressure of a graded component. We introduce C programming environments and practice using C compiling and debugging tools (makefiles, GDB, and Valgrind).

Swarthmore has a student mentoring program [18] for each of its introductory sequence courses, including CS 31. CS 31 student mentors primarily help students during weekly lab meetings, and during two weekly dedicated sessions that they run for CS 31-specific help. We occasionally use student mentors to help with some in-class group activities too.

III. CS31 Details

In a typical course schedule, CS 31 starts with binary data representation and then introduces C programming. Next, we introduce computer architecture and assembly. We then provide an overview of the memory hierarchy and the operating system. Finally, we cover shared memory parallelism, pthreads, and synchronization primitives. In this section, we discuss our focus and coverage of the main course topics and discuss details of the structure of the course.

A. Course Contents

C programming: CS 31 offers our students their first formal introduction to a systems programming language. We demonstrate C programming basics in the first two weeks of the course, and we gradually build students' competence by introducing new C semantics in tandem with the systems principles we cover each week. Our goal in CS 31 is not to provide an exhaustive C programming course, but rather, we emphasize the broader systems concepts and use C programming as a means to enable and further students' understanding.

We begin our C introduction by contrasting the high-level programming paradigms in C and Python, with topics such as explicit type declaration, differences between an interpreted and a compiled language, and memory management (allocating and freeing dynamic memory).

In introducing C, we initially describe variable scope, function syntax, representation of Boolean logic, primitive data types, conditionals, and loops. We also provide a preliminary introduction to composite data types (arrays, strings, and structs), their layout in memory, and performing basic I/O. Our introductory C lab assignment (Lab 2 in section III-B) introduces students to type declarations, C I/O and writing simple C functions.

As students get more proficient with C, we introduce a process's memory regions (the text, data, heap, and stack). We discuss the distinct features of each region and the OS's role in managing memory and ensuring the integrity of the stack and heap. Once we set the context of understanding program memory regions, we introduce pointers. We describe pointer types, `NULL` pointers, pointer declaration, initialization, assignment, and dereferencing. We introduce dynamic memory allocation, specifying the amount of memory requested, and freeing memory. This context gives us a natural segue into discussing C's philosophy of memory management, memory leaks, and segmentation violations. We aim to provide as many in-class and lab examples as

possible to familiarize students with understanding program memory regions, pointers, arrays, and string manipulation. We particularly emphasize the use of Valgrind for memory debugging. In some iterations of this course, we offer an additional string assignment (Lab 7), to help students gain more familiarity with using pointers, strings, and dynamic memory allocation.

In support of CS 31's PDC goals, we cover system calls (`fork`, `exec`, `wait`, and `exit`) and signals in conjunction with our introduction to operating systems. The details are described further under the sub-section Operating Systems. Once we introduce these concepts, we provide multiple opportunities for students to practice using these system calls during in-class examples and homeworks. We ask students to implement a Unix shell (Labs 8 and 9). Similarly, we introduce pthreads programming in the context of parallelizing applications, which we describe in more detail in the sub-sections Shared Memory Parallelism. For this section of the course, students implement two versions of Conway's Game of Life. In the first, the entire program runs serially (Lab 6), and in the second, students parallelize their earlier code using pthreads (Lab 10).

Binary Representation. Our first main "systems" topic is binary representation of C types and binary arithmetic. This is the first step in understanding how a C program is run by the underlying system. We focus primarily on C signed and unsigned values, 2s-complement encoding, addition, subtraction, and signed and unsigned overflow. We also discuss the typical number of bytes in different C types.

Finally, we discuss number representations and methods for converting between decimal, binary, and hexadecimal formats. We briefly discuss floating point representation, but do not expect students to be able to convert from binary to floating point. After covering binary representation, students have an understanding of the mechanics of binary arithmetic, memory sizes, and overflow. This background is helpful as we go on to cover computer architecture, caching, and paging later in the semester.

Architecture. Our coverage of architecture follows from the main course goal of understanding how a computer runs a program. We introduce the instruction set architecture (ISA) to help tie the architecture and program components together. Our coverage starts with the von Neumann architecture and showing that both instructions and data can be encoded in a binary representation and stored in memory. After a brief overview of the fetch, decode, execute and store cycle, we focus primarily on the design of the CPU and how it executes program instructions. Starting from basic AND, OR, and NOT logic gates, we design small circuits including arithmetic circuits like ripple carry adders, multiplexers, R-S latches, and gated D-latches. We stress abstraction along the way, building increasingly complex circuits from simpler ones. We use these components to discuss the design of a basic CPU consisting of an ALU and register file.

With prior introduction to binary representation, students begin to understand how individual bits can pass through circuits to perform computation, select from multiple inputs, and store results. We then add control circuitry, a program counter, and instruction registers to complete a simple CPU. We discuss instruction execution stages and how a clock circuit drives the execution. Along the way, we illustrate how instruction bits are used and how the CPU circuit is designed to execute the instruction stages on program data stored in CPU registers. Students gain practice through written homeworks and by building basic circuits and a complete simple CPU using Logisim [13] (Lab 3).

We also introduce instruction pipelining and multicore processors, presenting both in the context of improving efficiency. We discuss how pipelining makes efficient use of CPU circuitry resulting in an improved instructions per cycle rate. Our coverage of multicore highlights which CPU components are duplicated for each core and which are shared by cores. We also preview how operating systems make use of multicore computers to schedule multiple programs or parts of a single program to run simultaneously, in parallel, across the cores.

Assembly Programming. Our course also provides students with their first exposure to assembly language. We introduce assembly after students have learned how CPU circuitry is designed to run program instructions, including the stages of the execution pipeline and its clock-driven execution. Students also have had practice building a simple ALU for a small instruction set in Logisim, described in Lab 3 of section III-B. With this basic understanding of the underlying hardware, we next introduce the instruction set architecture (ISA) as the interface between the lowest software level and the hardware that executes it. We discuss the role of the compiler in translating a C program to the binary form, and assembly as the human-readable form of this binary machine code that the CPU understands. We currently teach 32-bit x86 assembly code because it represents a simplified form of the ISA of our lab machines and students can disassemble their own program binaries to the assembly code they learn.

We start with introducing the IA-32 register set and some basic arithmetic instructions, translating between C code and IA-32 equivalents, and stepping through their execution and the effects on registers and memory. Once students are familiar with the basics of assembly programming, we continue with assembly instructions for data movement, addressing modes, and changing control flow, tying these to C code examples with if-else, loops, function call/return, and stack memory. We continue to stress the relationship between C code examples and their IA-32 equivalents, translating between the two. Supporting function call and return and the stack is one of the most dense parts, requiring that students understand the effects of complex instruction execution on CPU registers and memory. We typically use

a full week to cover this part, and we give students lots of practice in class and on written homeworks. Our focus helps to reinforce program memory, scope, and function call and return. We additionally cover assembly translations for C code with arrays and pointers, and discuss memory layout of arrays. Students apply their understanding of assembly programming in written homework assignments, and in two lab assignments, Lab 4 where they translate C to IA-32 assembly that they compile and run, and Lab 5 where they use GDB assembly code tracing to discover the correct program input to find their way out of a binary maze program.

Our primary goal in teaching assembly is related to the main course goal of understanding how a computer runs a program. However, we also discuss some efficiency issues in the context of different equivalent assembly sequences and the effects of instruction memory accesses on performance.

Memory Hierarchy. We motivate our analysis of the memory hierarchy by describing the wide variety in performance characteristics (e.g., access latency, storage density, and cost) across storage devices. Given that there is no clear best type of memory for all scenarios, we discuss how a real system contains multiple memory technologies and that it needs to answer the question of *where* in-use program data should be stored at any given point in time to maximize performance.

Next, we introduce terminology for primary and secondary storage devices and describe the differences in their programming interfaces for accessing data (i.e., CPU instructions directly access primary storage via a memory bus versus a call to OS for secondary storage). We then introduce common storage devices that are likely to be found in a typical desktop or laptop system, characterize their performance trade-offs, and classify them as either primary or secondary.

We introduce the idea of a memory hierarchy with fast, low-density memory at the "top" and slow, high-density memory at the "bottom". To help students conceptualize how data moves through the hierarchy, we present a motivating group exercise that asks students where to store real-world physical objects (e.g., taking advantage of locality to store library books). This example builds students' intuition for using past accesses as a predictor for future behavior.

Armed with some real-world intuition, we formalize the notion of *locality* and differentiate how future access predictions might be either temporal or spatial in nature. We then perform another interactive exercise, this time with a block of C code, that asks students to identify several common instances of temporal and spatial locality in a program. At this point, students are ready to apply the general principles of the memory hierarchy and locality to cache systems.

Caching. We begin our coverage of caching by defining the cache conceptually as a faster subset of main memory. Using an abstract model of the cache, we introduce *hit* and *miss* terminology, starting with reading (loading) values from memory into the cache, and writing (storing) values back to memory. We also define cache lines and their basic metadata (valid and dirty bits). We build on this terminology to introduce cache writing policies, cache design (i.e., block size and associativity), and briefly analyze cache design trade-offs and their affect on the cache hit rate. Next, we introduce direct-mapped and set-associative cache designs to students.

Starting with a direct-mapped cache, we demonstrate examples of using the address of a memory operation to perform a cache lookup. As this is a frequent source of confusion for students, we pay particular attention to how various cache parameters like the block size and number of lines affect address division into the *tag*, *index*, and *offset*. Next, we introduce set-associative cache mechanics, primarily focusing on two-way set associativity to reduce complexity. We illustrate how associativity introduces a new challenge for eviction and the need for a replacement policy. While we ask students to brainstorm potential policies, we primarily concentrate on LRU, which connects to the locality intuition we built while covering the memory hierarchy. In our discussion of both cache designs, students work in small groups on several examples of cache accesses using a sequence of memory accesses (a mix of loads and stores).

We wrap up the discussion of caching by linking back to our initial introduction for the memory hierarchy and the ways in which data storage locations impact system performance. We present students with an interactive exercise in which two code blocks containing nested `for` loops access memory in different stride patterns. The exercise asks students to analyze their relative performance with cache behavior in mind.

Operating Systems. CS 31's coverage of operating systems primarily focuses on mechanisms and key abstractions, leaving most of the policies to our upper-level operating systems course. We begin the discussion by demystifying what an OS is and the role it plays as part of a computer system in efficiently managing hardware and providing an easy-to-use interface to users. As part of the demystification, we discuss a bit about how an OS boots onto the hardware and initializes itself to be prepared to run programs on the system.

We then introduce the process abstraction, motivating it as way for the OS to make efficient use of of hardware resources, linking this to our earlier coverage of the memory hierarchy and secondary storage devices. We introduce the idea of concurrency in the context of multiprogramming, timesharing, and process context switching.

Next, we discuss processes creation, introducing `fork` and the process hierarchy. We further emphasize concurrent execution after calls to `fork`. We also introduce the `exit`, `wait`, and `exec` system calls, with multiple code examples to illustrate their effects on the process hierarchy. The

examples highlight concurrent points in execution as well as dependencies or orderings of points in related processes executions (e.g., a parent process waiting for a child to terminate).

While describing processes, we also introduce asynchronous signals and the execution of signal handler code. We primarily focus on `SIGCHLD` signals as processes exit, but we also give students code examples of signal handlers for different types of signals to give them a feel for how asynchronous signals and signal handlers work. In conjunction with this material, we assign a lab where students implement a simple Unix shell program that includes support for running in the background and a signal handler for `SIGCHLD` signals. Our in-class and textbook code examples help to make this lab doable in a course at this level.

The second main OS abstraction we present is virtual memory (VM), and we note that students have already received instruction in CPU architecture, assembly programming, and caching prior to our introduction to virtual memory. We motivate virtual memory as a mechanism for implementing the process abstraction—each process is provided the same view of its own private address space while sharing physical RAM space, and the OS needs a mechanism to protect processes from accessing the contents of each others address space.

We also motivate VM as a means for the OS to make efficient use of RAM, allowing it to make memory appear to have larger capacity than physical RAM. We introduce single-level paged virtual memory and discuss virtual-to-physical address translation using a page table. We show examples of how page table mappings change on a context switch, page faults and page fault handling, LRU replacement, effective memory access time, and TLB caching of address translations to speed-up effective memory access time. This is another point where we emphasize concurrency in process context switching and how it affects page table mappings. We leave more advanced virtual memory topics, and a focus on policies, to our upper-level OS class.

Finally, we introduce parallelism immediately after our coverage of virtual memory. This provides a nice transition from OS to our primary coverage of shared memory parallelism by beginning with threads as another abstraction implemented by the OS.

Shared Memory Parallelism. Our last main topic covers shared memory parallel programming using pthreads. Having previously covered multicore architecture and the OS process abstraction, we begin by discussing the similarities and differences between threads and processes. Our explanation of threads, what threads share, and what they get private copies of (stack space and register values) follows easily from what students have just learned about processes, virtual memory, and the parts of a process's virtual address space. It also links back to the efficient use of hardware resources and revisits multicore processors. We introduce speedup and mention how resource contention can reduce observed speedup from theoretical ideal linear speedup in embarrassingly parallel applications.

We next introduce the pthreads library and show how to create, run, and join threads. We use some small examples, such as access to a shared counter, to introduce data races, critical sections, and atomic operations. In discussing synchronization primitives, we focus on the primitives provided by pthreads: mutex locks, barriers, and condition variables. We introduce the concept of Amdahl's law, but defer a deeper dive into this analysis to more advanced upper level courses. Once we introduce synchronization, we discuss the potential for deadlock, and we revisit speedup and using synchronization sparingly to enforce correctness while not having an overly large negative impact on performance. We finish the module with the producer/consumer (bounded buffer) problem, asking students to work in small groups to identify locations that require synchronization.

Our final lab exercise allows students to explore pthreads shared memory parallel programming by writing a pthreads implementation of Conway's game of life (Lab 10). A prior lab assignment has students write a sequential version of this application (Lab 6). The parallel version requires them to revisit and make some modest modifications to exploit data parallelism and process parts of a two-dimensional grid in parallel. A small amount of barrier and mutex synchronization is required to ensure correctness. The assignment allows students to compare correctness to their prior sequential solution while also allowing them to measure near linear speedup up to 16 threads on multicore machines.

Our choice to focus on shared memory parallelism as the first introduction parallelism follows from what students have learned in the course to this point. Our being able to rely on students already having learned about processes and virtual memory, concurrency, assembly, and machine organization with a bit about multicore processors, makes the introduction to parallelism, threads, shared memory, and synchronization follow as a natural "next step" in what they are learning. It also allows us to cover important parallel concepts in more depth and breadth than if students did not have this context. Finally, we specifically choose the approach of focusing only on shared memory parallelism as a way to develop parallel thinking skills more deeply by focusing on problems in just one parallel paradigm.

B. Labs and Homeworks

The CS 31 lab assignments cover a variety of topics, with increasing complexity of the course of the semester. For more details, a link to our recent offerings can be found at [5].

Lab 0, Tools for CS 31: Many students place out of our introductory course, so this warm-up lab covers basic Unix shell navigation and helps students set up their Swarthmore GitHub Enterprise accounts.

Lab 1, Data Representation and Arithmetic: This assignment consists of two parts. In the first, students answer written questions related to binary/hexadecimal number representation and arithmetic. In the second, students use simple C arithmetic statements to demonstrate properties of C variables (e.g., the maximum value that can be stored in an `int` variable).

Lab 2, C Programming Warm-up: Having developed basic C proficiency, students implement a basic $O(N^2)$ sorting algorithm that they already know from a CS1 course. This lab builds on students understanding of C type declarations, C I/O, and writing simple functions.

Lab 3 [24], Building an ALU Circuit: Students build, test, and simulate digital circuits using Logisim [13] They start by building small, standalone circuits (a sign extender and one-bit adder) and then combine them with additional logic to produce an ALU that supports eight operations and five status flags.

Lab 4, C Pointers and Assembly Code: Lab 4 contains two independent parts on C programming and assembly. The first requires students to compute basic statistics (mean, median, max) on input files with arrays of unknown types and sizes. Students learn to dynamically allocate and free memory, pass pointers, and write modular C functions. The second part of the lab involves writing short functions in assembly (e.g., swap two variables, or sum all values in an array). We also teach students the basics of Valgrind and GDB, and strongly encourage their use through the course.

Lab 5, Binary Maze: This lab is inspired by the "binary bomb lab" [4]. In this assignment, students work through a series of challenges ("floors" in a "maze") for which they use GDB to decipher assembly functions. Each floor requires a specific input pattern to advance. Each successive floor increases in complexity, giving students practice with understanding assembly and program control flow.

Lab 6, Game of Life: Students build a simulation for Conway's game of life. This lab introduces students to more complex memory allocation in the form of two-dimensional arrays for the game's grid. It also requires them to read game parameters and an initial grid state from a file. In addition to console output, we use the ParaVis [6] library to graphically visualize the simulation.

Lab 7, C String Library: After observing many students struggle with C strings in upper-level courses, we added this lab to CS 31. It requires students to implement and write test cases for several common C string library functions (e.g., `strcat`, `strcpy`, etc.). Note that Swarthmore's semesters have a different number of weeks in the fall and spring, so this lab is only offered when time permits.

Lab 8, Command Parser Library: Students practice building a library, with header files, to implement a command shell parser. The parser must tokenize a string and detect the presence of an ampersand character (indicating that the command should be run in the background).

Lab 9, Unix Shell: In lab 9, students build a shell that executes commands in the foreground and background. They use `fork` and `execvp` to start child processes and `waitpid` to reap terminated processes. We also require students to implement a simplified history mechanism.

In semesters that aren't able to teach the string lab, we often swap the order of labs 8 and 9, with the shell coming before the parser. In those semesters, we provide students with a parser library, which they replace with their own parser later.

Lab 10, Parallel Game of Life: Students extend their lab 6 simulation to execute on multiple threads in parallel using pthreads. Their solutions must partition the game grid vertically or horizontally, assigning responsibility for different regions to each of the threads. They use barriers to synchronize threads between rounds and a mutex to protect shared state. As in lab 6, we use the ParaVis [6] library to visualize the simulation, this time showing the thread regions in different colors. Visualizing the assignment in this way helps students to debug thread partitioning problems.

Written Homeworks. In addition to longer programming assignments, we assign weekly written homeworks. Homework assignments are short (1-2 hours to complete) and worth relatively little credit. Rather than assessing students, they promote student learning and help students prepare for exams by giving them extra practice on class topics. Each assignment focuses on applying a few key topics from the week's class content and readings.

Our current set of homeworks cover the following topics (assigned in the order listed):

- **C programming**: evaluating expressions, identifying types, function tracing, stack drawing;
- **Binary and arithmetic**: converting between decimal, binary, and hex, signed and unsigned arithmetic;
- **Circuits**: tracing through a circuit to produce its logic table, creating a circuit given a logic table;
- **C pointers**: type evaluation, code tracing, stack and heap memory drawing;
- **Simple assembly**: arithmetic instructions, showing memory and register contents, and converting to C;
- **Advanced assembly**: translate C conditionals and loops to assembly, trace assembly function calls, showing stack and register changes;
- **Direct mapped caching**: address division, tracing accesses showing hit, miss, and replacements on cache.
- **Set associative caching**: similar to direct mapped, applying LRU replacement;
- **Processes**: trace through C code examples with fork, exit, wait, draw process hierarchy, identify possible outputs from concurrent processes;
- **Virtual memory 1**: tracing through a single process's memory accesses using a page table, show effects on page table and RAM;
- **Virtual memory 2**: similar to VM1, but tracing through

two process' memory accesses, with context switching and LRU replacement;
- **Threads**: pthreads solution to producer/consumer started as an in-class exercise.

Over the years, we have experimented with different homework formats, including all-individual, individual or student-selected small groups, and assigning all students to homework and study groups that produce a single submission. We first used the assigned group approach during a remote COVID semester. Assigning all students to small study groups was designed to foster more group interaction with problem solving and exam preparation during a time when students where not meeting together in class. There was strong support for these groups from students, with a small number who did not find them helpful. From the faculty viewpoint, they were effective in fostering group learning. Additionally, their being assigned and required ensured that every student had at least one small group with which to collaborate. This model is something that we plan to consider for our in-person offerings as well.

IV. Evaluation

This section describes a preliminary evaluation of CS 31's efficacy in introducing the PDC concepts of Table I. We begin by highlighting some anecdotal evidence from CS 31 course evaluations by students across five recent offerings spanning from Spring 2018 to Spring 2020. Approximately 60 students take the course each semester, so these data represent responses from about 300 students total.

Several students appreciated their exposure to parallel programming concepts in CS 31. One student notes, *"I was exposed to a great deal of new concepts, especially in thinking about hardware, processors and new ways for managing programs like threading and forking*", while another student commented, *"Parallelism is great! Also pipelining. Those two concepts are super applicable to life broadly..."*. CS 31 has also increased student interest in taking upper-level systems courses, which students reporting sentiments such as, *"The ALU lab and learning how to parallelize programs were really interesting, and now I want to take parallel and distributed computing."* and *"I've always been interested in systems, and was extremely excited for this course. I'd say it exceeded my high expectations and I can't wait to take more systems courses!"*

A large number of student responses refer to a new systems perspective that CS 31 provided them — *"the course enabled me to learn about new concepts such as assembly code, parallelism, circuits, caches and other memory concepts", "whenever I create programs, I should keep in mind both [sic] BigO and what happens in hardware", "I like how we unpacked a lot of what goes on behind the scenes", and "It's cool to understand how the underlying hardware works"*.

Another goal of CS 31 is to build student confidence in thinking about system architectures, evaluating trade-offs, and writing performant, error-free programs. We share the following comments regarding student confidence after taking CS 31— *"I learned to be a more independent worker and gained confidence in my cs knowledge."*, *"The combination of theoretical approaches and practical application felt like a good mix ... the balance of the two really helped me learn."*. Another student shared, *"It was really interesting to think about the "give and take" nature or just questions of efficiency in different things we talked about, and see how those really apply to computer systems that we use every day. That kind of connection was really valuable and interesting to talk about, and it's something that isn't always present in other classes."*

We also collected student responses to open-ended questions from upper-level Operating Systems and Parallel and Distributed Computing courses that relate their understanding of PDC concepts back to their exposure to these concepts in CS 31. Students noted, *"I liked the application of all the different things I've learned in CS31 and CS35"*, *"CS31 got me very excited for and curious about computer systems. I was really excited to satisfy my deepened curiosity for operating systems after CS31 had sparked my interest."* and *"Ever since CS31, I wanted to learn more about how an OS works. I was interested in learning more about the mechanisms/policies and am interested in systems coding."*

In addition to quotes from student evaluations of CS 31, we collected survey data from two upper-level systems courses. In one course, *Parallel and Distributed Computing* (CS 87, Fall 2021), students took the survey at the end of a course as a reflection back on what they knew coming in. In the other, *Networking* (CS 43, Spring 2022), we administered the survey the first week of class, and we plan to run it again at the end of the semester as a post-course reflection.

The survey asked students to rate how well CS 31 prepared them for upper-level courses with PDC content. We presented students with a collection terms and concepts and a five-point rating scale based on Bloom's taxonomy. The ratings corresponded to: 0: do not recognize the topic/concept; 1: recognize the topic/concept/term; 2: could define it; 3: could analyze/understand this topic/concept in a solution that was given to me; and, 4: could apply this topic/concept to a problem.

Figure 1 shows the average and median values of students' ratings. These data show that, on average, students recognized all of these topics, and they feel comfortable explaining most of these topics as they start upper-level courses. For topics that CS 31 emphasizes heavily, such as the memory hierarchy, C programming, and some of the fundamentals of shared memory programming including race conditions, synchronization, and pthread programming, they rate their understanding at deeper levels.

Expected results are *not* all 4s for all of these topics.

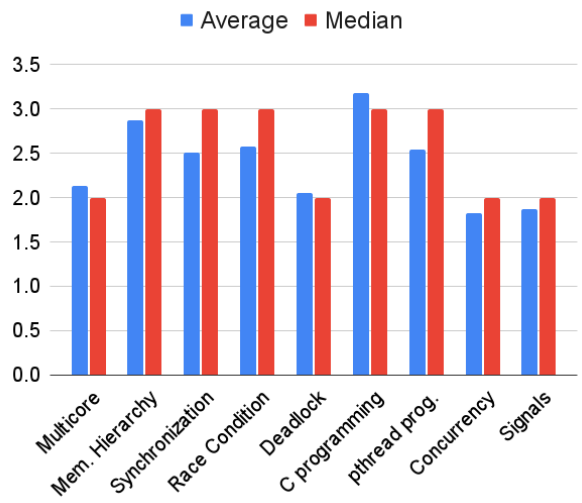

Figure 1: Upper-level students' rating of their understanding level of some PDC topics introduced in CS 31.

CS 31 is the first introduction to these concepts, and we do not cover all of them at the level where we would expect students to be able to apply them to a problem. However, we do expect that they should recognize all the terms and be ready to use them in upper-level courses, perhaps with a brief refresher. For some of the students surveyed, it has been up to two years since they took CS 31, and it is likely that their current understanding is lower than it would have been immediately after completing the course. We typically give "lab 0" assignments in our upper-level systems courses that are designed to get students back up to speed on CS 31 material and skills that they may not have used for awhile. We find student skill (and confidence in them) come back to students quickly after this practice.

In an open-ended question asking if they had any other comments when reflecting on CS 31, most students didn't add any comments. A few students said that they didn't remember much from CS 31 (it had been a while), and others in the post-course survey stated that they felt that CS 31 helped prepare them well for CS 87. One student noted, *"I think it gave me enough knowledge to feel comfortable taking CS 87"*, and another wrote, *"'I really liked how we started right where [CS] 31 left off."*

Finally, for the faculty who have taught upper-level courses with PDC content prior to our addition of CS 31, we see a substantial change in preparedness, knowledge, and comfort with PDC and systems concepts since adding CS 31 as a required part of our introductory sequence. With CS 31, students start significantly further ahead of where they were in the past. The background in computer systems and PDC that CS 31 provides results in students who are comfortable jumping into advanced work in these areas right away, and they naturally think in parallel and distributed ways from day one. Students come into our upper-level classes with programming and thinking skills that allow us to spend more time on more advanced PDC in our upper-level systems courses.

V. Related Work

Efforts to introduce PDC concepts early into the CS curriculum have to a large extent advocated the use of PDC modules in an existing CS1 [3], [7], [8] or a CS2 Data Structures course [2], [9], [10]. These PDC modules include unplugged activities [9], [14], [20] codifying canonical PDC paradigms, such as "patternlets" [2], [7] or adding one-to-two week PDC instruction modules [8], [10]. Our work in CS 31 is complementary to these efforts – it follows a philosophy of introducing PDC concepts more pervasively through the course, a philosophy shared by other pervasive PDC efforts described in [8], [11], [12].

Courses introducing PDC concepts have successfully used used Python [23], Java [3], [11] and C++ [2] with the OpenMP [8] and MPI libraries. A recent survey article provides detailed reports of use of different teaching paradigms and programming languages in use [21]. Our focus has been to introduce parallelism through C and to develop parallel thinking skills. We have found this to successfully translate to students' understanding of concepts of shared-memory, parallelism, and synchronization and contribute to their understanding of PDC concepts "ground-up". Our goal is to introduce parallelism pervasively through our curriculum [17] as this sets up students to also be exposed in upper-level courses to recognizing parallel concepts in other courses where they use CUDA, MPI, OpenMP, sockets, and python parallelization libraries.

VI. Conclusions

We presented the design, goals, and outcomes of an introductory-level computer systems course that introduces students to PDC topics, focusing on shared memory parallelism. Because our course requires only a CS1 prerequisite and it is itself a required prerequisite for upper-level courses, we ensure that all CS students are exposed to these important topics early. The course's curriculum introduces some PDC topics throughout, so end of semester coverage on shared memory parallel computing threads, and pthreads programming flows naturally from what students learned about computer systems along the way. We found that students learn and practice important parallel thinking skills that help prepare them for upper-level courses where they will use and expand these skills. The faculty teaching upper-level courses with PDC content note the benefits that this early exposure to PDC and parallel thinking provides our students as they move through our curriculum. All course materials, including homework and lab assignments, are available off the webpages [5] of our current and past offerings of the course.

References

[1] ACM/IEEE-CS Joint Task Force. Computer science curricula 2013. www.acm.org/education/CS2013-final-report.pdf, 2013.

[2] Joel C Adams. Injecting parallel computing into cs2. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 277–282, 2014.

[3] Steven A Bogaerts. One step at a time: Parallelism in an introductory programming course. *Journal of Parallel and Distributed Computing*, 105:4–17, 2017.

[4] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[5] CS31: Introduction To Computer Systems Course Webpages. http://www.cs.swarthmore.edu/~newhall/cs31.

[6] Andrew Danner, Tia Newhall, and Kevin Webb. A Library for Visualizing and Debugging Parallel Applications. In *Proc. of 9th NSF/TCPP Workshop on Parallel and Distributed Education (EduPar-19)*, 2019.

[7] Russell Feldhausen, Scott Bell, and Daniel Andresen. Minimum time, maximum effect: Introducing parallel computing in cs0 and stem outreach activities using scratch. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, pages 1–7, 2014.

[8] Sheikh Ghafoor, David W Brown, and Mike Rogers. Integrating parallel computing in introductory programming classes: an experience and lessons learned. In *European Conference on Parallel Processing*, pages 216–226. Springer, 2017.

[9] Sheikh K. Ghafoor, David W. Brown, Mike Rogers, and Thomas Hines. Unplugged activities to introduce parallel computing in introductory programming classes: An experience report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education ITiCSE'19*, 2019.

[10] Dan Grossman and Ruth E. Anderson. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12. Association for Computing Machinery, 2012.

[11] Max Grossman, Maha Aziz, Heng Chi, Anant Tibrewal, Shams Imam, and Vivek Sarkar. Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. *J. Parallel Distrib. Comput.*, 105(C), July 2017.

[12] David J. John and Stan J. Thomas. Parallel and distributed computing across the computer science curriculum. In *Parallel and Distributed Processing Symposium Workshops IPDPSW, 2014 IEEE International*, May 2014.

[13] Logisim. https://github.com/Logisim-Ita/Logisim.

[14] Suzanne Matthews. PDCunplugged: A free repository of unplugged parallel and distributed computing activities. In *2020 IEEE/ACM Workshop on Parallel and Distributed Computing Education (EduPar-20)*, 2020.

[15] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. Dive into Systems. https://diveintosystems.org/, 2020.

[16] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. Dive into systems: A free, online textbook for introducing computer systems. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE'21)*, 2021.

[17] Tia Newhall, Andrew Danner, and Kevin C. Webb. Pervasive parallel and distributed computing in a liberal arts college curriculum. In *Journal of Parallel and Distributed Computing*, volume 105, July 2017.

[18] Tia Newhall, Lisa Meeden, Andrew Danner, Ameet Soni, Frances Ruiz, and Richard Wicentowski. A support program for introductory cs courses that improves student performance and retains students from underrepresented groups. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 2014.

[19] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. A Multi-Institutional Study of Peer Instruction in Introductory Computing. In *ACM Technical Symposium on Computing Science Education (SIGCSE)*, 2016.

[20] S. Srivastava, M. Smith, A. Ghimire, and S. 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)*, 2019.

[21] 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.

[22] The NSF/IEEE-TCPP Curriculum Working Group. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates, version 2.0. https://tcpp.cs.gsu.edu/curriculum/?q=home, 2020.

[23] Neftali Watkinson, Aniket Shivam, Alexandru Nicolau, and Alexander Veidenbaum. Teaching parallel computing and dependence analysis with python. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 320–325. IEEE, 2019.

[24] Richard Wicentowski. CPU logisim lab assignment, 2008.