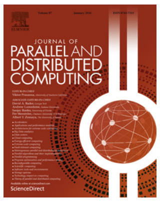
J. Parallel Distrib. Comput. 105 (2017) 53–62

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc



Pervasive parallel and distributed computing in a liberal arts college curriculum

Tia Newhall*, Andrew Danner, Kevin C. Webb

Computer Science Department, Swarthmore College, Swarthmore, PA, USA

HIGHLIGHTS

- We describe a CS undergraduate curriculum that incorporates PDC topics throughout.
- A new intro sequence course that includes PDC.
- PDC expanded or added into 9 upper-level course.
- Tailored to a liberal arts college, but applicable broadly.

ARTICLE INFO

Article history:
Received 14 June 2016
Received in revised form
31 December 2016
Accepted 5 January 2017
Available online 14 January 2017

Keywords:
CS curriculum
Parallel and distributed computing

ABSTRACT

We present a model for incorporating parallel and distributed computing (PDC) throughout an undergraduate CS curriculum. Our curriculum is designed to introduce students early to parallel and distributed computing topics and to expose students to these topics repeatedly in the context of a wide variety of CS courses. The key to our approach is the development of a required intermediate-level course that serves as an introduction to computer systems and parallel computing. It serves as a requirement for every CS major and minor and is a prerequisite to upper-level courses that expand on parallel and distributed computing topics in different contexts. With the addition of this new course, we are able to easily make room in upper-level courses to add and expand parallel and distributed computing topics. The goal of our curricular design is to ensure that every graduating CS major has exposure to parallel and distributed computing, with both a breadth and depth of coverage. Our curriculum is particularly designed for the constraints of a small liberal arts college, however, much of its ideas and its design are applicable to any undergraduate CS curriculum.

1. Introduction

"*The past decade has brought explosive growth in multiprocessor computing, including multi-core processors and distributed data centers. As a result, parallel and distributed computing has moved from a largely elective topic to become more of a core component of undergraduate computing curricula*". [1]

Instruction in parallel and distributed computing has traditionally been relegated to a few isolated courses, taught primarily in the context of scientific computing, distributed systems or computer networks. With the ubiquity of multi-core CPUs, GPUs, and clusters, parallel systems are now the norm. Furthermore, the era of Big Data and data-intensive computing has ushered in an expansive growth in the application and use of parallel and distributed computing. These two trends together have led to parallel and distributed computing becoming pervasive throughout computer science, resulting in their increasingly becoming a core part of the field.

The ubiquity of parallel and distributed computing is also reflected in the ACM/IEEE Task Force's 2013 CS education curriculum [1] that added a new knowledge area in Parallel and Distributed Computing, which stresses the importance of teaching parallel computation throughout the undergraduate curriculum. Additionally, the NSF/IEEE-TCPP 2012 Curriculum Initiative on Parallel and Distributed Computing [17] provides guidance and support for departments looking to expand the coverage of parallel and distributed topics in their undergraduate programs.[1]

Prior to our curricular changes, we taught parallel and distributed computing in only two of our upper-level elective

* Corresponding author.
E-mail addresses: newhall@cs.swarthmore.edu (T. Newhall), adanner@cs.swarthmore.edu (A. Danner), kwebb@cs.swarthmore.edu (K.C. Webb).

[1] The changes to our curriculum were partially supported by the TCPP Early Adopters program.

http://dx.doi.org/10.1016/j.jpdc.2017.01.005
0743-7315/© 2017 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

courses. As a result, many of our CS majors had no instruction in these topics. The changes we made were driven in part by our desire to ensure that every Swarthmore CS major and minor is exposed to parallel and distributed computing.

There are several main goals in the design of our curriculum:

1. Ensure that students are exposed to parallelism early by integrating it into our introductory sequence.
2. Provide repetition of this content so that students are exposed to parallel and distributed topics multiple times.
3. Provide both a breadth of topic coverage as well as opportunities for students to go in depth in some areas.
4. Expose students to parallel and distributed topics in the context of multiple sub-disciplines rather than being isolated into specialized parallel and distributed courses. We want our curriculum to mirror the ubiquity of parallel and distributed computing by integrating these topics into a broad range of courses across our curriculum.

In addition to our primary goals, we also want our efforts to increase opportunities for students to participate in parallel and distributed research projects.

Ultimately, we want every student to be exposed to fundamental issues in parallel and distributed computing from the algorithmic, systems, architecture, programming, and applications perspectives. Our pedagogical focus is to teach students the skills to analyze and problem solve in parallel and distributed environments; our overriding focus is on teaching "parallel thinking."

In Fall 2012 we first introduced changes to our curriculum that were designed to meet these goals. Our solution had to work within the constraints of a small liberal arts college, most notably, we could not increase the number of required courses for the major or deepen the prerequisite hierarchy of our classes.

The key component of our curricular change is the addition of a new intermediate-level course, Introduction to Computer Systems. It covers machine organization, an introduction to operating systems, and an introduction to parallel computing focusing on shared memory parallelism. The addition of this new course allowed us to factor out introductory material from many upper-level courses, leaving space in these classes that we could easily fill with new and expanded parallel and distributed computing content.

To date, we have added and expanded coverage of parallel and distributed computing in eight upper-level courses. We continue this expansion both within courses that already have some content and also into courses that traditionally have not had such coverage. Prior to our curricular changes students could graduate with a CS major from Swarthmore without ever being exposed to computer systems or to parallel and distributed computing. Since our change, every graduating CS major and minor has both breadth and depth of exposure to these important topics.

2. Background

Before describing our current curriculum in depth, we present institutional context for our curricular changes and describe our departmental constraints. Swarthmore is a small, elite liberal arts college with approximately 1600 undergraduate students. The Computer Science Department consists of seven tenure track faculty and offers CS major and minor degrees. Our curriculum is designed to balance several factors, including the small size of our department, the expertise of our faculty, and the role of a computer science curriculum in the context of a liberal arts college [12]. Our pedagogical methods include a mix of lectures, active in-class exercises, and labs. Many of our graduates eventually go on to top CS graduate schools; for this reason, our curriculum includes a focus on preparing students for graduate study by providing them instruction and practice in reading and discussing CS research papers, technical writing, oral presentation, and independent research projects.

The overall goal of our curriculum is to increase proficiency in computational thinking and practice. We believe this will help both majors and non-majors in any further educational or career endeavor. We teach students to think like computer scientists by teaching algorithmic problem solving, developing their analytical thinking skills, teaching them the theoretical basis of our discipline, and giving them practice applying the theory to solve real-world problems. We feel that by teaching students how to learn CS, they master the tools necessary to adapt to our rapidly changing discipline.

The nature of a liberal arts college poses several challenges to expanding parallel and distributed content in our curriculum. Typically, liberal arts colleges require that students take a large number of courses outside of their major. At Swarthmore, students must take 20 of the 32 courses required for graduation outside of their major.

Because of our small size, we are not able to cover all areas of computer science (programming languages is one example for which we do not currently have a tenure-track expert). We provide an introductory sequence of three core courses and a set of upper-level electives designed to provide depth and breadth to students. Individual upper-level courses are usually only offered once every other year, which means that a student may have only one opportunity to take a specific course. It also means that our courses need to be tailored to accommodate a wide variety of student backgrounds—in any given upper-level course there can be senior CS majors alongside underclassmen taking their very first advanced CS course.

These constraints dictate that our CS major cannot include a large number of requirements, that we need to provide several course path options for students to satisfy the major, and that we need to have a shallow prerequisite hierarchy to our courses. In both our old and our new curriculum we have just three levels in our course hierarchy: an introductory course; two intermediate-level courses; and upper-level courses that require only our intermediate-level courses as prerequisites.

2.1. *Our curriculum prior to 2012*

Prior to 2012, we had a much smaller department with four tenure lines. Our curriculum at the time included three introductory sequence courses: a CS1 course taught in Python; a CS2 course taught in Java prior to 2010 and C++ after 2010; and an optional Machine Organization course that included an introduction to C programming. Because of the constraints of being in a liberal arts setting and our course frequency, all of our upper-level courses only had CS1 and CS2 as prerequisites. After taking CS2, students needed to take one of Theory of Computation or Algorithms, one of Programming Languages or Compilers, one of Machine Organization or Computer Architecture, our senior seminar course, and three upper-level electives. We also required two math courses beyond second semester Calculus.

The first half of the Machine Organization course covered binary data representation, digital logic structures, ISA, assembly language, and I/O. The second half was an introduction to C programming for students who had already completed a CS1 course. The Computer Architecture course was taught by the Engineering Department at Swarthmore, and followed a typical undergraduate-level Computer Architecture curriculum. Neither of these courses included computer systems topics, nor parallel and distributed computing topics. In addition, because these classes were not prerequisites to upper-level CS courses, we could not rely on students

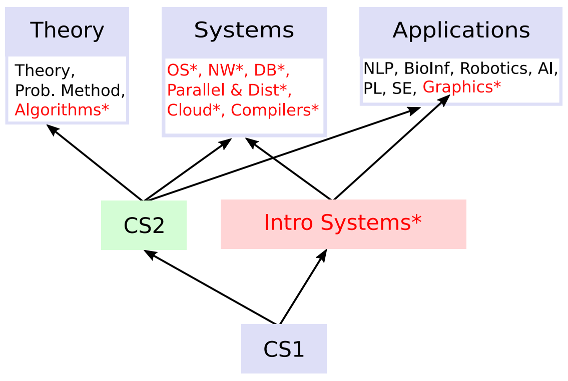

Theory
Theory, Prob. Method, Algorithms*
Systems
OS*, NW*, DB*, Parallel & Dist*, Cloud*, Compilers*
Applications
NLP, BioInf, Robotics, AI, PL, SE, Graphics*
CS2
Intro Systems*
CS1

Fig. 1. Our new curriculum design showing the prerequisite hierarchy. The newly added Introduction to Systems course is at the intermediate-level and is a prerequisite to about 1/2 of our upper-level courses (arrows). Starred (and in red) are courses with PDC topics. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

having seen any machine organization or computer architecture content in our upper-level courses.

Our previous introductory sequence prepared students well in algorithmic problem solving, programming, and algorithmic analysis, and thus prepared students well for about one half of our upper-level courses. However, we found that their lack of computer systems background made them less prepared for many of our upper-level courses in systems-related areas. As a result, we had to spend time in each of these courses teaching introductory systems material and C programming. These courses seemed more difficult to the students new to this material, while being repetitive to students who had seen this material in other upper-level courses. Repeating introductory material also frequently forced us to cut advanced material.

2.2. Our new curriculum

In Fall 2012 we first introduced changes to our curriculum designed to meet our goals of adding and expanding parallel and distributed computing topics. There are two main parts of our curricular changes [5]: a new intermediate-level course that first introduces parallelism and changes to upper-level requirements to ensure that all students see advanced parallel and distributed computing topics [8]. Our new prerequisite structure is depicted in Fig. 1.

The key component of our curricular change is the addition of a new intermediate-level course, Introduction to Computer Systems. It replaces our Machine Organization course, serves as the first introduction to parallel computing, and ensures that all students have a basic computer systems background to prepare them for upper-level systems courses. Its only prerequisite is our CS1 course (Introduction to Computing), and it can be taken before, after, or concurrently with our CS2 course (Data Structures and Algorithms).

One extremely useful side-effect of our adding this new course is that it resulted in making space in our upper-level courses into which we could easily add and expand parallel and distributed computing coverage. Before the addition of this class, it was necessary to teach introductory systems and C programming in every upper-level systems course. Typically, this introductory material accounted for between two to three weeks of these courses, and it could not be covered in as much depth or breadth as it can in our new course, which has an entire semester to devote to these topics. With the addition of Introduction to Systems as a new prerequisite, all students now enter upper-level CS courses with instruction in C, assembly programming, computer systems, architecture, and parallel computing. This gives us 2–3 weeks that we can use to add in parallel or distributed computing topics.

Table 1
The names and revision dates of the PDC courses involved in our new curriculum.

Course	Revision date
Introduction to computer systems (CS31)	Fall 2012 (New course)
Operating systems (CS45)	Spring 2014
Computer networks (CS43)	Fall 2013 (New course)
Parallel and distributed computing (CS87)	Spring 2016
Cloud systems and data center networks (CS89)	Fall 2014 (New course)
Database systems (CS44)	Fall 2016
Compilers (CS75)	Spring 2017
Algorithms (CS41)	Fall 2012
Graphics (CS40)	Spring 2013

The second component of our curricular change is the grouping of upper-level courses into three main categories (Theory, Systems, and Applications) with a requirement that students take at least one upper-level course in each group. Because we added parallel and distributed computing content to every Systems course and to courses in the other groups, every CS major now sees advanced parallel and distributed computing content in a variety of contexts. The courses containing parallel and distributed content are starred in the list below:

- **Group 1, Theory and Algorithms**: Algorithms*, Theory, The Probabilistic Method.
- **Group 2, Systems**: Networking*, Databases*, Operating Systems*, Compilers*, Cloud Systems and Data Center Networks*, Parallel and Distributed Computing*.
- **Group 3, Applications**: Graphics*, AI, Natural Language Processing, Information Retrieval, BioInformatics, Software Engineering, Adaptive Robotics, Programming Languages.

After completing the intro sequence and group requirements, students must take two additional upper-level electives, including any from the three groups, or Computer Architecture, Mobile Robotics, or Computer Vision, which are taught by our Engineering department. We also require a senior seminar course, and two math courses beyond second semester Calculus.

3. Curriculum and courses

Towards our goal of broadly exposing students to parallel and distributed computing, we revised our curriculum in 2012 to incorporate most of the NSF/IEEE-TCPP recommendations. This is an on-going effort, involving at least nine courses, three of which are new, while the others are existing courses to which parallel topics have been added or expanded. The set of courses, each of which is described in detail throughout the remainder of this section, and their revision dates are displayed in Table 1. Links to course web pages are available in Appendix A, and Appendix B provides a complete list of the NSF/IEEE-TCPP topics covered in each course.

3.1. Introduction to computer systems (CS31)

Central to our curricular redesign is CS31, a new "Introduction to Computer Systems" course, which serves as a prerequisite to our revised upper-division courses. CS31 is designed to be a next course after our introductory course, so we must ensure that students entering CS31 having taken only our CS1 course are not overwhelmed by the content or by the C programming assignments. It has become a required course for all CS majors and minors since being first offered in Fall 2012. We emphasize three overarching topics in CS31:

- How a program goes from being expressed in a high-level programming language to being executed on a computer.

Table 2
Topics covered in CS31.

Primary topic	Details
The memory hierarchy	Storage circuits, RAM, Disk, Caching and Cache Organizations, Paging, Replacement Policies, Cache Coherence
Multicore and threads	Architecture, Buses, Coherency, Explicit parallelism, Threads and threaded programming
Operating systems	Overview, Goals, Processes, Threads, Synchronization primitives, Virtual memory, Efficiency, Mechanism/Policy and Space/Time Trade-offs
Parallel algorithms and programming	Shared memory, Threads, Synchronization, Deadlock, Race conditions, Critical sections, Producer–Consumer, Amdahl's Law, Scalability, Parallel speed-up
Other in-depth topics	Machine organization, Assembly programming, C to IA32, The stack, Function call mechanics
Other high-level topics	Distributed computing, Message passing basics, TCP/IP Sockets, Pipelining, Super-scalar, Implicit parallelism

Table 3
A typical CS31 course schedule.

Week	Topic(s)	Lab assignment
1	Data representation, C Language	Binary conversion, Arithmetic
2	Binary arithmetic	C Warmup
3	Digital circuits	Logic simulator: ALU
4	ISAs, Assembly	
5	Pointers, Memory	C Pointer and assembly exercises
6	Functions and the stack	Assembly debugging puzzles
7	2D Arrays, Structs, Strings	
8	Memory hierarchy	Game of life
9	Caching	Strings
10	Operating systems, Processes	Shell
11	Virtual memory	
12	Concurrency, Threads	Parallel game of life
13	Synchronization, Deadlock	

- The systems-level costs associated with program performance and how to evaluate trade-offs in system design.
- Parallel computing, including algorithms and systems programming, with a focus on shared memory parallelism and threaded programming.

Secondary course goals include: learning C, assembly, and pthreads programming; learning debugging and debugging tools such as gdb and valgrind; designing and carrying out performance experiments; and working collaboratively in small groups.

CS31 includes many topics from the TCPP curriculum, with a focus on covering the minimal skill set. Topics covered span the Architecture, Programming, and Algorithms areas of the TCPP curriculum. Table 2 describes CS31's core topics, and Appendix B provides a detailed list of the course's TCPP topics.

The course is structured as a vertical slice through a computer, with the goal of providing students a complete picture of how computers operate. Table 3 outlines a typical CS31 semester schedule. As this is a broadly-focused course, most topics are covered for approximately one week, with the notable exception of concurrency and parallel programming getting two. For each topic, we illustrate the core concepts and describe the commonly used abstractions or models. We generally defer covering complex details, in-depth analyses, and less commonly used alternatives to upper level courses; CS31 aims to provide students the background to study such topics in depth.

The first half of the course emphasizes low-level building blocks like binary data representation, circuits, assembly, and memory organization. The remaining weeks highlight software support structures including program compilation and execution, core operating system abstractions, memory performance, and parallel programming. Throughout, it serves our students as a first introduction to machine organization and computer systems, parallel architectures, and systems programming. We underscore the importance of analyzing problems from a systems perspective. For example, students use what they learn about the memory hierarchy to evaluate the performance of code based on its memory costs in addition to its algorithmic complexity.

CS courses at Swarthmore include an associated lab section that meets weekly for 90 min. The labs are used to teach students the programming tools necessary for carrying out lab assignments, to provide practice on lecture content, and to help students with their lab work. The lab assignments engage students with a practical application of the topics covered in lecture. In CS31, lab assignments consider both sequential programs written in C and parallel programs running on multi-core computers using pthreads.

CS31 replaced a previous course in our curriculum on machine organization. The old course was not a prerequisite to any of our upper-level courses. It also was one of two options for satisfying a requirement for the major (computer architecture being the other option), thus not all CS majors took machine organization, and we could not rely on students in upper-level courses knowing either machine organization or computer architecture.

CS31 covers approximately the same machine organization topics as our previous course, but additionally includes about half of a semester of computer systems and parallel computing topics that were not previously covered by the machine organization course. We were able to create a little space for these new topics in CS31 by reducing slightly the coverage of the digital logic level and the amount of time devoted to assembly language programming. However, CS31 spends roughly the same amount of time on machine organization topics as we did in our previous machine organization course.

We were able to add coverage of systems and parallel computing topics into CS31 by restructuring the way in which we teach C programming. CS31 labs teach students C in the context of machine organization, systems, and parallel computing topics, whereas Machine Organization taught C programming as a standalone topic. By teaching C alongside the primary course topics, we are able to add about one half of a semester's worth of new topics on systems and parallel computing into CS31, without losing much coverage of machine organization.

Overall, we are very pleased with CS31. Our end-of-course surveys show that students enjoy the course and feel that they take away a lot from the experience. Thus far, it seems to be preparing our students well for upper-division systems courses.

3.2. *Operating systems (CS45)*

Prior to the addition of CS31, our Operating Systems course began with an introduction to C programming and C programming tools and a quick introduction to computer systems and computer architecture, including an overview of operating systems and an introduction to the memory hierarchy. The course then proceeded to follow a fairly standard undergraduate OS curriculum, covering processes and threads, scheduling, synchronization, memory management, file systems, I/O, protection and security, and finally some advanced topics when time permitted.

Course projects involve changes to the Linux kernel. Students develop test suites to test the correctness of their kernel changes. The course strives to have a good balance of theory and practice. It includes a strong focus on analyzing performance based on systems costs, trade-offs in system design, abstraction and layered design, and the separation of mechanism and policy.

With the addition of CS31 as a new prerequisite, we were able to replace the first couple weeks of introduction to C, systems, and architecture with advanced systems topics, including both more breadth and depth of coverage of parallel and distributed computing. In addition to already knowing something about systems, students come into OS having already learned about shared memory parallelism and threads. They also have already written pthread programs and solved synchronization problems including implementing a basic critical section with a mutex, using barrier synchronization, and solving the bounded buffer producer–consumer problem. Because students now enter OS with this background, we can cover synchronization more quickly and also in more depth then in previous versions of the OS course.

The new version of OS covers a fair amount of parallel and distributed computing topics intertwined in the coverage of the standard OS topics. For example, when teaching process and thread scheduling, we include scheduling for multi-core and parallel systems, discussing affinity and gang scheduling. When teaching virtual memory, we describe mechanisms for threads to share address spaces, and we discuss issues associated with caching shared addresses and false sharing. In covering synchronization, the additional background from CS31 in the architecture of multicore and of the memory hierarchy and caching mechanisms allows for more in-depth discussion of implementing synchronization primitives and discussing trade-offs in blocking vs. spinning.

In the most recent offering of the course, we added an introduction to distributed systems, using distributed file systems as an example. We compared centralized, decentralized and peer-to-peer design while discussing trade-offs in these designs and how the use and scale of the filesystem leads to different design choices. We also were able to include an introduction to network computing, focusing on TCP/IP sockets. When teaching security topics, students were able to discuss issues in more depth because they had a stronger distributed and network computing background than in the past.

Overall, expanding coverage of parallel and distributed computing into this course was easy. There are numerous places in a typical OS curriculum where parallel and distributed topics can be added or expanded, and we found it to be natural to integrate parallel and distributed computing throughout this course. The benefit of this type of integration is that students are thinking about parallelism and concurrency in every OS subsystem, which results in more practice in developing parallel thinking skills than when parallel and distributed topics are relegated to the end of the class as "advanced topics". We observed that by the end of the class, students naturally thought about problems related to concurrency and parallelism.

3.3. Computer networks (CS43)

In our Networking course, students explore the underpinnings of digital communication, with an emphasis on the modern Internet. By the end of the course, students are expected to design and evaluate network protocols, analyze the separation of design concerns into abstraction layers, and construct applications with complex communication requirements.

CS43's class lectures cover parallel and distributed topics like message passing, asynchronous communication, decentralized routing protocols, peer-to-peer protocols, and distributed hash tables. We frequently ask students to analyze the outcome of hypothetical scenarios involving several concurrently-operating entities during in-class exercises and exams.

The course's lab assignments, primarily written in C, force students to grapple with parallel and distributed computing topics in several contexts. Early in the course, students build a multithreaded web server that must be capable of serving multiple concurrent client requests. Later, they design and implement a protocol for streaming MP3 audio across the network. Like the web server, their audio server must handle multiple client connections, but we require non-blocking I/O (*select*) as opposed to threads. The audio client creates independent threads to enable human interaction, data retrieval, and audio playing to occur simultaneously. Finally, the students design a TCP-like reliable message delivery protocol that manages message retransmissions in the face of asynchrony and message loss. Throughout the lab portion of the course, we highlight the importance of designing an application prior to implementation and underscore the adoption of socket programming and error checking best practices.

CS43 leans heavily on our CS31 prerequisite, expecting that students are already familiar with C programming and threads. Course projects expand on these topics by examining server concurrency models (e.g., thread-per-client vs. non-blocking I/O) in depth. Furthermore, the CS31 background enables us to assume that students are familiar with common shared memory computing models like the producer–consumer problem. Such assumptions allow the course to pursue significantly more ambitious programming projects than would otherwise be possible with students who are seeing concurrency for the first time.

3.4. Parallel and distributed computing (CS87)

Parallel and Distributed Computing was added to our curriculum in 2010, replacing a course in Distributed Systems. CS87 is a broad survey of parallel and distributed computing. It is organized as a combination lecture and seminar-style course. Students read and discuss research papers, and propose and carry out independent projects during the second half of the course. The lectures cover design issues in the context of parallel systems, parallel algorithms, and distributed systems.

The learning goals for the course emphasize paper reading, discussion, writing, oral presentation, and critical analysis, as well experimentation, testing, and proposing and carrying out an independent research project.

The first half of the course provides exposure to a variety of parallel and distributed programming models through several short lab assignments. One purpose of this breadth of coverage is to teach students different tools that they may choose to use in their independent course project. The short labs include a focus on API design, experimental design, and testing, with the purpose of helping to prepare students for the course project.

Prior to the introduction of CS31 into our curriculum, CS87 often was students' first introduction to computer systems and was the lone parallel and distributed computing course in our curriculum.

With the introduction of CS31 as a prerequisite, all students now come into CS87 knowing C programming, computer systems, caching and the memory hierarchy. They also have background in parallel computing, including pthread programming and solving some synchronization problems. With this common preparation, we are able to remove introductory C and pthreads programming assignments from CS87 to make room for more breadth in the assigned labs, as well as to add more emphasis on testing and experimentation.

In the most recent offering of the course, the short labs included: parallel matrix multiply in OpenMP; implementing Game of Life in pthreads and designing and carrying out a scalability study of the solution; implementing parallel Odd–Even sort in MPI and testing its performance for large-size problems on TACC's Stampede cluster [16], available through XSEDE [14]; implementing a parallel fire simulator in CUDA that included experimentation of different CUDA block and thread layouts; and a C socket application that implemented a multi-party talk client–server application. This project also asks students design an API for a voting protocol for joining an existing conversation. Students must test their solutions with those of other students for compatibility.

The course topics cover a large percentage of the TCPP and the ACM curricular recommendations. For example, the most recent offering included: shared memory systems, GPUs, MPPs, clusters, P2P, grid and cloud computing, SIMD, MIMD, data parallel, client–server, distributed memory, shared memory, threads, synchronization, MPI [13], CUDA [15], OpenMP [7], Map-Reduce [9], hybrid CPU–GPU-MPI programming, distributed communication, parallel programming patterns, parallel reduce and scan, tradeoffs, speed-up, scalability, dependencies, time, power, parallel algorithms, fault tolerance, distributed file systems, distributed shared memory, security, and networking.

Along with the breadth of coverage of parallel and distributed computing topics, students get depth of exposure, most notably through the independent course project. The overriding goal of the course project is to give students a taste of what it is like to do research. The course project consists of three main phases. In the first phase, students find and refine a project topic organized around a general problem to solve. They develop a solution to this problem and a plan for implementing and evaluating their solution. This step requires that students review related research work. At the end of the first phase, students submit a written project proposal and an annotated bibliography. The second phase consists of a mid-way project report and oral presentation to the class. The last phase consists of a final written report that is structured like a research paper and an oral presentation of their project, much like a CS conference presentation. The project is not required to be novel research, but often times it is. Several projects started in this class have lead to CS research publications for students.

3.5. Cloud systems and data center networks (CS89)

Cloud Systems is a new course that has been taught once (Fall 2014) in a seminar style, with class meetings focusing on the discussion of primary research literature. It combines advanced topics at the intersection of Operating Systems, Networks, Distributed Systems, and Databases. The primary goals for the course are to explore the composition of modern large-scale cloud services and to teach students how to read and evaluate published research papers.

The course aims to strike a balance between theoretical and applied topics. Initially, the course examines seminal papers, with students reading about the individual technologies behind cloud computing, such as virtualization and large-scale network topologies. As the course progresses, we shift the focus towards more recent advances in cloud infrastructure, like data centers, replicated state machine algorithms, scalable data processing systems (e.g., MapReduce), and cloud-based distributed storage applications. Finally, we tie the topics together by holistically analyzing real cloud platforms as described by major providers (e.g., Google, Facebook, and Amazon). Many students reported that they particularly enjoy reading about the details of cloud services that they personally use on a regular basis.

Students in CS89 are expected to complete four lab assignments and a student-selected final project in pairs. While some labs focus more on network infrastructure, the first and fourth labs highlight distributed computing in the forms of a MapReduce-based movie recommendation system and a distributed chat server in which students must achieve consistency in the face of unexpected message orderings. Many students also choose to work on distributed systems in their final projects, by further exploring MapReduce, implementing Paxos, or developing other related topics. Thanks to a generous grant from Amazon Web Services (AWS) [3], students gain experience with real cloud services like Amazon's Elastic Compute Cloud and Elastic MapReduce to perform their lab assignments and final projects.

3.6. Database systems (CS44)

Our Databases course focuses on database management systems. It includes in-depth coverage of the design and implementation of database management systems, relational and ER database models, query languages, indexes, and query optimization. The main course projects include implementing parts of a database management system (DBMS). Students implement different layers of a system including a low-level memory buffer manager, a Heap File for storing records, and a B+Tree index. They also implement relational database operations including select, project, and merge and indexed join methods into their DBMS. Several smaller projects give students practice in relational and ER database design and in using SQL.

Prior to adding CS31 into our curriculum, the first couple of weeks of this course was devoted to teaching students C/C++ programming and programming tools, an introduction to systems, data storage, and the memory hierarchy. Students needed this background to implement the course projects and as background context for the discussion of course topics.

With the addition of CS31, students enter with a much stronger C programming background and a solid foundation in computer systems and the memory hierarchy. As a result, we gain at least two weeks that was previously devoted to teaching students this material. This allows us to expand coverage of more advanced Database topics, including the addition of new parallel and distributed database content. Our main focus has been introducing these topics in the context of scaling a DBMS to support very large databases. We have added an introduction to distributed database systems that includes discussion of partitioning and replication, consistency, distributed locking and transactions, and scalability and fault tolerance. We also cover parallel algorithms for implementing relational operations, focusing on parallel join methods. Other ideas for the future include adding a discussion of distributed hash tables, MapReduce, relaxed consistency constraints, and read-optimized databases.

3.7. Compilers (CS75)

Our Compilers course follows a fairly common undergraduate compilers curriculum, including detailed coverage of parts of the front-end and back-end of a compiler. Students implement a compiler for most of the C programming language, implementing a lexical analyzer, parser, and a code generator with some optimizations, in a multi-part semester-long project.

Prior to the addition of CS31, the Compilers course also served as an introduction to C programming and to C and Unix programming tools. It provided background in computer architecture, and was the only coverage of assembly language and assembly language programming in our curriculum. Similar to the other upper-level systems courses, several weeks of the course were devoted to getting students up to speed on these topics.

With the addition of CS31 as a new prerequisite, our students now come into Compilers with extensive C and assembly programming experience. In particular, they have traced and written IA32 assembly code for function calls, stack operations, branches and loops, and array and struct memory layout. In-class exercises and exams show students have a good understanding of scope and the difference between stack, heap and global memory. As a result, we now have room in this course to expand the coverage of compiler optimization and to add advanced compiler topics. Although we have not yet taught the new version of our Compilers course, our plan is to incorporate additional parallel content. We will include a discussion of compiler optimizations for super-scalar and multi-core systems. We also will cover automatic parallelizing compilers, with a focus on the goals and difficulties of implementing automatic parallelization. We also plan to discuss incremental parallelism, focusing on generating parallel loop code based on the fork-join model of parallelization, using OpenMP as an example.

3.8. Algorithms (CS41)

Our upper-level algorithms course is in our Theory group. The course explores the fundamentals of algorithmic design and analysis, including transforming abstract problem descriptions into a formal algorithmic statement and developing algorithmic solutions to these problems. We emphasize proving correctness and analyzing both runtime and space complexity.

While we primarily use Kleinberg and Tardos [11] for the textbook, we sample parallel material from Cormen et al. [6]. To smoothly blend parallel topics into a traditional algorithms course, we start with the traditional merge sort algorithm and analyze its complexity in alternative models of computation including the parallel models (PRAM, BSP/CILK) and the external memory (out-of-core, or I/O-efficient) model. Since students have the greatest classroom exposure to the traditional RAM model and the $O(n \log n)$ merge sort bound, we examine how to measure complexity in other models. Students see that algorithmic techniques from earlier in the semester including asymptotic analysis are still relevant, even though alternative computation models might not focus on the total number of computation steps. We use similar techniques to analyze parallel topics of work and span, or bound the number of memory transfers in the I/O model. We introduce the notion of speedup and scalability and provide a number of in-class exercises and exam questions that allow students to explore parallel complexity through a theoretical lens.

We cover alternate models of computation in two weeks of class time. An end of course evaluation in Fall 2012 specifically asked students to comment on alternate models of computation including the I/O model and parallel models. Feedback was generally positive and several students commented on how these models seem more applicable to current computational problems. Some students mentioned that the material on I/O-efficient and parallel algorithms was covered too quickly, and that they wished there was more coverage of parallel algorithms. In the future, we may expand the coverage into a third week or eliminate the I/O-efficient model to allow more time for parallel models.

3.9. Computer graphics (CS40)

Computer Graphics is an upper-level course in our group of Applications courses. The primary focus for the course is on data structures and algorithms for representing and rendering 3D models. Lab assignments use C++ and OpenGL 3.x to implement core Graphics concepts including modeling hierarchies, camera transforms, and complex lighting. We also introduce parallel topics through general purpose GPU (GPGPU) computing using CUDA.

Instead of presenting CUDA and GPGPU computing as topics completely disjoint from traditional computer graphics material, the focus of modern OpenGL on shader-based programming makes the transition to GPGPU computing easier. In this model, developers write small shader programs that manipulate graphics data in parallel in a SIMD fashion. Each modern OpenGL application usually consists of at least two shader programs: a *vertex* shader and a *fragment* shader. The vertex shader runs in parallel on each geometric point in the scene in parallel, while fragment shaders run on each potential output pixel in parallel.

By introducing common shader programs early and explaining a little about what goes on behind the scenes, students quickly learn that the GPU is programmable and that shaders are optimized to run on the highly parallel hardware of the GPU. We then gradually replace the geometric data processed by the vertex shaders with a general buffer of values and manipulate those buffers using CUDA *kernels*, which essentially replace the role of the graphics shaders. Our introduction to CUDA uses some basic examples including vector addition, dot products, and parallel reductions. We then spend a week on parallel algorithms and synchronization primitives: map, filter, scatter, gather, and reduce. In a third week, we tie CUDA concepts back to core graphics concepts by using CUDA to manipulate images typically with fractal generation or fast image processing filters.

Since CS31 is now a prerequisite, students have some prior background with parallel programming using pthreads and are familiar with memory address spaces and caching. Using shaders in the beginning of the course, we can quickly introduce additional basic PDC topics including SIMD and stream architectures and memory organization (CPU memory, GPU memory). We introduce more advanced topics of GPU threads, synchronization, parallel GPU core scheduling, and parallel speedup later when we dive into CUDA and peek behind the scenes of the GPU architecture.

Student response to CUDA and PDC concepts has generally been positive. While gaining a deep understanding of the power and capabilities of CUDA in three weeks, a few student groups have used CUDA a part of their final projects including building a GPU ray-tracer and modeling a complex dynamical system. Other students used pthreads to accelerate their midterm project of building a CPU-only ray-tracer. Several students have expressed excitement in watching applications achieve linear speedup over 32, 128, or 512 cores and beyond. The highly parallel architecture of modern GPUs allows students to extend parallelism beyond the small number of CPU cores, and exposes performance bottlenecks when certain algorithms are not designed to leverage all GPU cores.

4. Evaluation

The introduction of CS31 was part of a larger curriculum overhaul, so it is difficult to provide a direct assessment CS31's impact compared to the our prior curriculum. One primary outcome of our new curriculum is that students see parallel and distributed computing topics at the introductory level and in greater depth in at least one other upper level course. We guarantee advanced coverage of parallel computing material through our new requirement that all students take at least one upper-division Systems courses, each of which contains parallel and distributed computing topics.

Without expanding the number of courses required for the major, adding CS31 as a new introductory requirement forced us to give up one course from the old major. Thus, we cut the number of elective courses required for the major from three to two, and we no longer offer a Machine Organization course. We continue to cover machine organization topics in CS31. Students interested in exploring these topics can still take computer architecture in the Engineering Department and count their study toward elective credit in the CS major.

Our new curriculum and CS31 requirement brings significant gain to our upper-level systems courses. Since we can assume background material from CS31 in our systems courses, we can go into greater depth in systems topic, or cover advanced topics that were completely skipped in upper level courses prior to CS31. We typically gain two to three weeks of advanced material in each of these courses as a result of having CS31.

Anecdotal evidence suggests students are indeed "thinking in parallel" in these upper level courses. Prior to CS31, students would not ask questions about e.g., parallel schedulers for OS processes, or if parallelism can be used to accelerate a task. These questions have become more common as students see parallel computing topics in more contexts. In graphics, several students have explored parallel GPU computing topics as part of their final projects.

Overall, our changes have improved the depth and quality of our systems courses without sacrificing quality outside of the systems area. We feel our model can be used as an example for other departments considering integrating parallel and distributed computing topics without needing to make significant changes or cuts to their existing curriculum.

5. Conclusions

We continue to expand and enhance our coverage of parallel and distributed computing topics throughout our curriculum. Our efforts began in Fall 2012 with the introduction of a new intermediate course, CS31: Introduction to Computer Systems. Motivated by the NSF/IEEE-TCPP 2012 Curriculum Initiative on Parallel and Distributed Computing [17], we strive to encourage parallel thinking across multiple courses. Through recent curricular changes to our major and minor programs, all CS students will graduate with experience in parallel thinking. Our expanded coverage of parallel and distributed computing topics now spans three new courses, five courses modified to include PDC topics, and planned modifications for the next offering of our compilers course.

In many cases, we have found we can integrate parallel computing topics without sacrificing core content. The addition of CS31 allows us to present common background material for all our systems courses and frees up time to explore advanced topics, including parallel topics, in our upper level courses. Additionally, CS31 provides an opportunity to introduce parallel computing early in the curriculum, ensuring all CS students can begin "parallel thinking" early in their studies. By incorporating parallel content in a variety of courses, including algorithmic, programming, systems, and applications courses, parallel computing topics are no longer isolated in a single special topics elective course and thus become a more familiar approach to solving computational problems. Students can also explore parallel topics across a breadth of computer science areas, and go further in depth in systems courses with extensive parallel and distributed computing content.

As a relatively small CS department at a liberal arts college, we are limited in the number and frequency of our course offerings. We found that we could guarantee practice with parallel computing topics early by adding a new required course focused on introduction to computer systems. To address the limitation of infrequent upper level course, we found we could distribute parallel and distributing computing concepts throughout multiple upper level courses across multiple sub-disciplines of computer science. By starting small and leveraging the expertise of our faculty, we hope our efforts can be used by other institutions looking to introduce or expand parallel computing content in their departments. We have begun to share the course materials we developed with colleagues at other CS departments who are interested in adopting this material. See Appendix A for links to these syllabi, lectures, and lab assignments. Our work complements other related efforts to support parallel and distributing computing education [4,2,10].

Overall, we feel our initial implementation and evaluation of our curricular changes are a success. We plan to continue enhancing, adding, and integrating parallel topics throughout the curriculum so students have an opportunity to take courses with parallel and distributed topics every semester. Through parallel thinking in multiple courses, students are better prepared for academic research or opportunities in industry using parallel computing topics.

Appendix A. Course webpages

The following are urls to example webpages for the new versions of the courses described in this paper. A web page containing all of these links can be found at www.cs.swarthmore.edu/~adanner/jpdc.

- Introduction to Computer Systems (CS31): www.cs.swarthmore.edu/~newhall/cs31
- Operating Systems (CS45): www.cs.swarthmore.edu/~newhall/cs45
- Computer Networks (CS43): www.cs.swarthmore.edu/~kwebb/cs43
- Parallel and Distributed Computing (CS87): www.cs.swarthmore.edu/~newhall/cs87
- Cloud Systems and Data Center Networks (CS89): www.cs.swarthmore.edu/~kwebb/cs91[2]
- Database Management Systems (CS44): www.cs.swarthmore.edu/~newhall/cs44
- Algorithms (CS41): www.cs.swarthmore.edu/~adanner/cs41
- Graphics (CS40): www.cs.swarthmore.edu/~adanner/cs40
- Compilers (CS75): www.cs.swarthmore.edu/~newhall/cs75.[3]

Appendix B. NSF/IEEE-TCPP curriculum topics by course

Primary topic	Details
CS31: Introduction to computer systems	
Taxonomy	Multicore, Superscalar
Memory hierarchy	Cache organization, Atomicity, Coherence, False sharing, Impact on software
Parallel programming	Shared memory, Message passing, Task/thread spawning
Semantics and correctness	Tasks and threads, Synchronization, Critical regions, Producer–consumer, Concurrency defects, Deadlocks, Race conditions
Performance issues	Data distribution, Data layout, Data locality, False sharing, Speedup, Efficiency, Amdahl's law
Parallel and distributed models and complexity	Time, Space/memory, Speedup, Cost tradeoffs
Cross-cutting topics	Locality, Concurrency, Non-determinism
CS 45: Operating systems	
Classes	Shared vs. Distributed memory, SMP, Message passing, Bandwidth, Packet-switching
Memory hierarchy	Atomicity, Consistency, False sharing, Impact on software
Parallel programming Paradigms and notations	Shared memory, Distributed memory, Message passing, Client–Server, Task/thread spawning
Semantics and correctness Issues	Tasks and threads, Synchronization, Critical regions, Producer–Consumer, Monitors, Concurrency defects, Deadlocks, Data races
Performance issues	Scheduling, Data layout, Data locality, False sharing, Performance, Performance metrics, Amdahl's law
Parallel and distributed models and complexity	Time, Space/memory, Speedup, Cost tradeoffs, Dependencies
Algorithmic problems	Communication, Synchronization
Cross-cutting topics	Locality, Concurrency, Power consumption, Fault tolerance, Performance modeling
Current/advanced topics	Cluster computing, Security in distributed systems, Performance modeling
CS 43: Computer networks	
Message passing	Topologies, Routing, Packet switching, Circuit switching, Latency, Bandwidth
Parallel programming	Shared memory, Client/Server, Task/thread spawning
Semantics and correctness	Tasks and threads, Synchronization, Deadlocks, Race conditions

(continued on next page)

[2] This course was renumbered. The home page still refers to "CS91."

[3] The current link is a version prior to our adding CS31 as a pre-req, and thus does not contain parallel or distributed content. When we update this course to the new version with parallel and distributed content, a link to it will be added here.

Primary topic	Details
Algorithmic problems	Broadcast, Synchronization, Asynchrony, Path selection
Current/Advanced topics	Peer to peer computing, Web services
CS 87: Parallel and distributed computing	
Classes	Taxonomy, ILP, SIMD, MIMD, SMT, Multicore, Heterogeneous, SMP, Buses, NUMA, Topologies, Latency, Bandwidth, Packet-switching
Memory hierarchy	Cache organizations, Atomicity, Consistency, Coherence, False sharing, Impact on software
Performance metrics	Benchmarks, LinPack
Parallel programming Paradigms and notations	SIMD, Shared memory, Language extensions, Compiler directives, Libraries, Distributed memory, Message passing, Client–Server, Hybrid, Task/thread spawning, SPMD, Data parallel, Parallel loops, Data parallel for distributed memory
Semantics and correctness Issues	Tasks and threads, Synchronization, Critical regions, Producer–Consumer, Concurrency defects, Deadlocks, Data races, Memory models, Sequential and relaxed consistency
Performance issues	Computation, Commutation decomposition strategies, Program transformations, Load balancing, Scheduling mapping, Data, Data distribution, Data layout, Data locality, False sharing, Performance, Performance metrics, Speedup, Efficiency, Amdahl's law, Gustafson's Law, Isoefficiency
Parallel and distributed models and complexity	Asymptotics, Time, Space/Memory, Speedup, Cost trade-offs, Scalability in algorithms and architectures, Model-based notions, CILK, Dependencies, Task graphs
Algorithmic paradigms	Divide and conquer, Recursion, Scan, Reduction, Dependencies, Blocking, Out-of-core algorithms
Algorithmic problems	Communication, Broadcast, Multicast, Scatter/gather, Asynchrony, Synchronization, Sorting, Selection, Specialized computations, Matrix computations
High level themes	What and why is parallel/distributed computing
Cross-cutting topics	Locality, Concurrency, Non-determinism, Power consumption, Fault tolerance
Current/advanced topics	Cluster computing, Cloud/Grid, Peer-to-Peer, Security in distributed systems, Performance modeling, Web services
CS 89: Cloud systems and data center networks	
Memory hierarchy	Atomicity, Consistency, Coherence, Impact on software
Parallel programming	Message passing, Client/Server
Semantics and correctness	Sequential consistency, Relaxed consistency
Performance issues	Load balancing, Scheduling and mapping, Data distribution, Data locality
Algorithmic paradigms	Reduction (MapReduce)
Algorithmic problems	Broadcast, Multicast, Asynchrony
Cross-cutting topics	Concurrency, Locality, Fault tolerance
Current/Advanced topics	Cluster computing, Cloud computing, Consistency in distributed transactions, Security in distributed systems, Peer to peer computing
CS 44: Database systems	
Classes	Shared vs. Distributed memory, Message passing
Memory hierarchy	Atomicity, Consistency, Impact on software
Parallel programming Paradigms and notations	Shared memory, Distributed memory, Message passing, Client–Server, Task/thread spawning
Semantics and correctness Issues	Tasks and threads, Synchronization, Critical regions, Concurrency defects, Deadlocks, Data races, Sequential consistency
Performance issues	Scheduling, Data locality, Data distribution, Performance
Parallel and distributed models and complexity	Time, Space/Memory, Speedup, Cost tradeoffs, Dependencies
Algorithmic paradigms	Divide and conquer

Primary topic	Details
Algorithmic problems	Communication, Synchronization, Sorting, Selection
High level themes	What and why is parallel/distributed computing
Cross-cutting topics	Locality, Concurrency, Fault tolerance
Current/advanced topics	Consistency in distributed transactions, Security in distributed systems
CS 75: Compilers	
Classes	ILP, SIMD, Pipelines
Memory hierarchy	Atomicity, Consistency, Coherence, False sharing, Impact on software
Performance metrics	CPI
Parallel programming Paradigms and notations	SIMD, Shared memory, Language extensions, Compiler directives, Task/thread spawning, Parallel loops
Semantics and correctness Issues	Tasks and threads, Synchronization, Critical regions, Sequential consistency
Performance issues	Program transformations, Load balancing, Scheduling, Static and dynamic, False sharing, Monitoring tools
Parallel and distributed models and complexity	Time, Space/Memory, Cost tradeoffs, Dependencies
Algorithmic problems	Synchronization, Convolutions
CS 41: Algorithms	
Parallel and distributed models and complexity	Asymptotic bounds, Time, Memory, Space, Scalability, PRAM, Task graphs, Work, Span
Algorithmic paradigms	Divide and conquer, Recursion, Reduction, Out-of-core (I/O-efficient) algorithms
Algorithmic problems	Sorting, Selection, Matrix computation
Cross-cutting topics	Locality, Concurrency
CS 40: Graphics	
Classes	SIMD, Streams, GPU, Latency, Bandwidth
Parallel programming Paradigms	Hybrid
Semantics and Correctness Issues	Tasks, Threads, Synchronization
Performance	Computation decomposition, Data layout, Data locality, Speedup
Algorithmic problems	Scatter/gather, Selection
Cross-cutting topics	Locality, Concurrency

References

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

[2] J. Adams, R. Brown, E. Shoop, Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to CS undergraduates, in: Parallel and Distributed Processing Symposium Workshops and Ph.D. Forum IPDPSW, 2013 IEEE 27th International, 2013.

[3] Amazon, Amazon Web Services (AWS), http://aws.amazon.com/.

[4] C.M. Brown, Y.-H. Lu, S. Midkiff, Introducing parallel programming in undergraduate curriculum, in: Parallel and Distributed Processing Symposium Workshops and Ph.D. Forum IPDPSW, 2013 IEEE 27th International, 2013.

[5] Computer Science Department, Swarthmore College, Swarthmore computer science department curriculum, 2012. http://www.swarthmore.edu/cc_computerscience.xml.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., The MIT Press, 2009.

[7] L. Dagum, R. Menon, OpenMP: and industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (2002).

[8] A. Danner, T. Newhall, Integrating parallel and distributed computing topics into an undergraduate cs curriculum, in: Proc. Workshop on Parallel and Distributed Computing Education, 2013. URL http://www.cs.swarthmore.edu/~adanner/docs/eduPar13UndergradParallelism.pdf.

[9] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, USENIX Association, 2004.

[10] D.J. John, S.J. Thomas, Parallel and distributed computing across the computer science curriculum, in: Parallel and Distributed Processing Symposium Workshops IPDPSW, 2014 IEEE International, 2014.

[11] J. Kleinberg, E. Tardos, Algorithm Design, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[12] LACS Consortium, A 2007 model curriculum for a liberal arts degree in computer science, J. Ed. Res. Comput. (JERIC) 7 (2007).

[13] E. Lusk, Programming with MPI on clusters, in: 3rd IEEE International Conference on Cluster Computing, CLUSTER'01, 2001.
[14] National Science Foundation grant number OCI-1053575, XSEDE Extreme Science and Engineering Discovery Environment, 2011. http://www.xsede.org.
[15] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture, 2016. http://www.nvidia.com/object/cuda_home_new.html.
[16] Texas Advanced Computing Center (TACC), Stampede Supercomputer, 2015. https://www.tacc.utexas.edu/stampede.
[17] S.K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, J. Wu, NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates, Version I, 2012. http://www.cs.gsu.edu/~tcpp/curriculum/index.php.

Tia Newhall is a professor in the Computer Science Department at Swarthmore College. She received her Ph.D. from the University of Wisconsin in 1999. Her research interests lie in parallel and distributed systems focusing on cluster storage systems.

Andrew Danner has B.S. degrees in Mathematics and Physics from Gettysburg College and a Ph.D. in Computer Science from Duke University. He is currently an associate professor of Computer Science at Swarthmore College. His research interests include external memory and parallel algorithms for Geographic Information Systems.

Kevin C. Webb received the B.S. degree in Computer Science from the Georgia Institute of Technology in 2007 and Ph.D. degree from the University of California, San Diego in 2013. He is currently an assistant professor in the Computer Science department at Swarthmore College. His research interests include computer networks, distributed systems, and computer science education.