

Breadth in Depth

A 1st Year Introduction to Parallel Programming

Thomas R. Gross
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
thomas.gross@inf.ethz.ch

ABSTRACT

There is a debate on when and how to teach parallel programming in the undergraduate curriculum. We try to structure the debate along a number of dimensions and then present the solution that we adopted for an engineering-oriented curriculum. We added an introduction to parallel programming to the list of mandatory classes in the 2nd semester. The class exposes students to three styles of parallel programming: threads with shared memory, CSP-style message passing, and OpenMP-based parallel programming. Within these models, the class aims to focus the student's attention on communication as the key issue in parallel programs. Explicit communication (or their absence) causes correctness problems, implicit communication (e.g., when accessing shared data in different threads) forces the student to understand when updates are globally visible. An introductory class early in the undergraduate curriculum has a number of benefits and disadvantages, which are discussed in this paper. A preliminary evaluation after two editions of this course indicates that the design goals are met but also points to several issues that other institutions may want to consider.

Categories and Subject Descriptors: K.3.2 [Computer and Information Science Education] Computer Science Education; D.1.3 [Concurrent Programming] Parallel programming

General Terms: Design, Experience

Keywords: CS education, parallel programming, software engineering

1. INTRODUCTION

Many papers, panels, and keynote presentations have repeated that current and future computing systems are parallel computers. Computer science students must learn not only how to construct a sequential program but must also learn how to master parallel programming. As a result many institutions have included mandatory or elective courses on

parallel (or concurrent) programming[16, 3, 19] in the undergraduate curriculum or have made special efforts to introduce parallelism concepts into the existing courses[7, 4].

But whereas many educators and practitioners agree that parallel programming is important and should be included, there is a lot less agreement on the topics that should be thought, if the material should be part of a separate course (or courses), and when to position the course(s) on this topic in the curriculum. The questions that were posed at the (first) panel on Parallel Computing Education at Supercomputing'91 still have not found a universally accepted answer[17]. This paper does not provide a definite answer either (nor is it likely that there will be a definite answer anytime soon, if ever) but attempts to add to the discussion by presenting another approach that has been implemented in an engineering-oriented computer science curriculum. We start with the rationale and discuss the concerns.

1.1 Options

When debating what parallel programming topics to teach and when to teach them within the context of an undergraduate curriculum there are a number of questions to address. What do we expect the student to gain from attending a class? Of course none of these questions have only two answers that one must choose from - often the realities of an institution's curriculum require a compromise.

1. What is the *target audience*? Who are the customers of the course? All Computer Science majors? Bachelor or Masters students? Future application experts, e.g., in a program on Computational Engineering and Science? How close to graduation are the students?
2. What is the *target platform*? Is there a specific parallel system (either because your institution provides access to this system or you anticipate that this kind of platform will be most important for your students)? Are there specific tools or programming languages that are important for other classes?
3. What should be the *course content and scope*? How many hours of lecture and lab can be tolerated (or are expected) by your audience? How can you balance coverage of principles and an exposition to current tools or current practice?

I try to address some of these questions below. The answer to the last question (what is the scope in credits or hours) is specific to any institution. To provide some frame

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

Copyright 2011 ACM 978-1-4503-0500-6/11/03 ...\$10.00.

of reference I assume that – regardless of level and target audience – such a course on parallel programming should take up approximately 1/5 of the students time. Approximately means that in some institutions it may be a 1/4; i.e. the course could be one of four or five courses taken by a student in a semester. Within the context of the European Credit Transfer System (ECTS), such a course yields about 6 - 8 credits. (The ECTS system defines a full load for a semester to be 30 credits.) In practice, such a course would include about 180 min of lectures per week in a semester of about 14 weeks, with ample opportunities (and requirements) to participate in labs, work out homework assignments, and time for practical programming. No doubt, it is possible to design a good class on parallel programming with more or fewer credits (and, therefore, higher or lower expectations on the time a student can spend on this topic).

1.2 Target audience

If the participants are Computer Science students, the emphasis should be on principles, with more emphasis on principles the earlier the class is placed in the curriculum. There exist today many Computer Science programs that differ significantly in the kind (and depth) of topics that are included in the curriculum. It would be surprising if these different programs all agreed on the contents of a class on parallel programming. Programs with an emphasis on computer systems or software engineering may include a class that is different from a class that serves programs with emphasis on visual computing and graphics or human-computer interaction. If the class audience consists of students that will use parallel systems as a platform for application development then the principles and abstractions that are covered in a class on parallel programming must be reinforced by appropriate application-oriented techniques and tools[14].

1.3 Target platform

The proliferation of multicore (and soon manycore) processor components provides ample motivation to pursue parallel programming as such components are used in many systems. On one hand this situation presents a unique opportunity: if students own a laptop (many or all of them do at many institutions) then they own and operate a parallel computer. These computers are small-scale symmetric shared-memory multiprocessor (there are 2 to 4 cores to execute independent instruction streams, each core has access to all memory cells, and the access time is roughly the same for all cores and all memory cells). The abundance of parallel computers at the students' disposal is great as it allows students to experience first-hand the problems of parallel computing. Even a simple parallel program that is incorrect has a good chance to expose the bug on the student's system and therefore motivates students to learn about parallel programming. So a course with this situation in mind might emphasize a simple view of parallel systems (different execution units with a shared, easily accessible memory system).

But there are two problems with this story: once the number of cores increases, the computer system architecture may provide the programming abstraction of a shared memory but the actual computer system implementation presents non-uniform access times. I.e. the computer system is a NUMA (Non-Uniform Memory Architecture) system, and programmers ignore the distance between execution units

and the storage system at their peril. Even today's 8-core systems are no longer symmetric with regard to memory access time or memory bandwidth, and larger systems will likely be even more asymmetric. The other issue is related to the other end of the spectrum: if you consider large-scale parallel systems, with tens of thousands of cores (or more), then the programming model used on these platforms is message passing: MPI [10] is the de-facto standard of programming high-end parallel systems.

Graphics processors (and their friends like Cell-based systems) provide another option, and a number of groups have used CUDA [6] as a platform for parallel computing[16]. These systems also provide easy access to a large(r) number of cores with impressive peak performance. Unfortunately the programming support for these system is low-level with limited portability, even from one generation of graphics processors to the next.

1.4 Course content

When designing a class, it is never easy to find a balance between theory and practice, regardless of subject area. To complicate the discussion in the context of parallel programming, there are multiple dimensions of theory (parallel algorithms, formal models of parallel computing, ...) and a wide range of practical issues (computer architecture, programming languages, libraries, performance monitors, tuning tools, ...) that an instructor can choose to include or ignore.

The wide range of programming languages, libraries, and tools complicates the situation tremendously, and in many cases, there is a steep learning curve. To complicate matters further, some popular tools (e.g., CUDA, OpenMP [5], MPI) are only loosely connected to a theoretical foundation, and new tools and languages are introduced frequently. Students must gain experience in parallel programming, but it is not wise to devote a significant part of a semester to learn a tool or programming language that is never used again. On the other hand, any practical (lab-oriented) component of a class on parallel programming is bound to use some tool, with its associated overhead.

2. BREADTH-FIRST AT A BASIC LEVEL

ETH decided to include a mandatory class on parallel programming early in the curriculum. Students starting in the fall of 2008 or later must take the "Introduction to Parallel Programming" in their 2nd semester. There exist several other classes for more advanced students (seniors, or seniors/masters). Some of these courses concentrate on specific platforms (e.g., MPI for computational scientists, MapReduce for information systems students), others cover parallel algorithms or multiprocessor programming based on the later chapters on the book by Herlihy and Shavit[11]. By offering a mandatory course early in the curriculum, the department wanted to ensure that all students are exposed to the problems of parallel processing¹.

The overall goal of the class is to teach the students how to write correct parallel programs To structure the class and

¹The depressing story of the Therac-25 incidents[15], and the slow progress in eliminating similar problems in software development, should serve as a warning: a computer science graduate should at least identify potential data races and call in an expert if he or she is not skilled in programming a parallel system.

to expose the student to the wide range of options in parallel programming we use three widely-used programming models as guiding principle. We look at multithreading with shared memory as an example of programming with explicit and implicit communication steps. A program may use synchronization to force an ordering (or to avoid a data race) or may cause implicit communication as a consequence of accessing an object shared between multiple threads. We explore message passing as an extreme approach to allow only explicit communication and allow the students to experience the benefits and pain of this programming model. Finally, we use OpenMP as an approach that combines user hints with runtime scheduling. Students experience how some computations can be parallelized with a few hints, how difficult it is in other cases to determine if such hints are legal, and how much time can be spent on uncovering the consequences of a faulty hint. The focus on programming models also appears to be a reasonable preparation for a later discussion of parallel patterns[13].

Students should understand the important role of communication (implicit communication through assignments to shared data, explicit communication by sending or receiving messages). To appreciate the difficulties associated with the use of shared data, the students must delve into variable assignments so that a discussion of update visibility is possible. To understand the dangers of circular waiting in message passing, the students must understand that data transfers incur time. The desire to teach these issues led to the “in depth” exposure of the various programming styles.

But there was also another goal for an early class on parallel programming: the curriculum provides the students with many options and as a consequence, there are not many unifying themes that students can use to organize the vast amount of material that a student is expected to master. Parallelism will be a critical issue for many future software development efforts. And, as has been noted by others, various aspects of parallelism are often introduced in other courses. A class on operating systems may cover semaphores and threads, a class on databases may cover similar concepts, and classes on networks or computer architecture may also devote a significant part of their lectures to discuss threading and its variants. So by moving the basic concepts to a single class, we allow on one hand other courses to focus on more advanced topics and at the same time allow the student to realize that the concepts (or at least some of them) are useful in many contexts.

We therefore designed a class that provides breadth (a decision made by others in a similar situation as well[16]) but we chose a different level. Rather than exploring different programming languages, we want students to understand the core issues of parallel programming. So the class aims to provide breadth at a rather deep level. It is important that students understand that communication is a key problem and there exist several options to express communication (or to trigger communication).

2.1 Platform issues

To keep the faculty and the students sane, we try to minimize the overhead that is associated with introducing new languages - ideally, students can explore these topics within the context of a single programming language. As a target system, we use the student’s laptop, although we provide also laboratories for those students that do not own or

carry a laptop. These laboratories contain dual-core systems (dual-boot Windows and Linux) as well as a smaller number of 4-core and 8-core systems (Linux).

2.2 Non-topics

The flip side of using the student’s computers is that it is difficult (often impossible) to see any performance improvement due to multiprocessing. With only two cores available, it is often the case that a parallel program is slower than the sequential one. While this is a worthwhile learning experience, this situation does not make it easy to motivate students to embrace parallel systems. We address this situation in three different ways: we make available – later in the course – larger systems with multiple cores. We look for application programs that show some speedup (the poor performance is often not due to parallelism overhead but due to memory system effects; if multiple tasks or threads create cache conflicts, the benefits of parallelism are overshadowed by slowdowns due to memory traffic). Finally, we sidestep performance optimization and delay the in-depth discussion of optimizations and tuning to later semesters.

Performance optimization is clearly important for parallel programming, and it has been argued that this topic should be in an integral part of any class on parallel programming[8]. Clearly the motivation for parallel computing is to obtain a faster solution than can be provided by a sequential system (or to obtain a better solution than could be obtained on a sequential system). But covering performance optimization requires that students know a fair bit about computer architecture and possibly about compilers, as the details of the memory organization and the optimizations chosen by the compiler have a great influence on application performance.

In light of the early nature of this class, we do not cover all aspects of the memory model for multithreaded systems[1]. Our goal is to teach the students to write correct programs, not programs that can be optimized well or that use performance-oriented functions or methods. E.g., in the context of Java, we rely only on the simple methods for classes in a package like `java.util.concurrent.atomic` and hide the “weak” methods from the students. Since we do not cover performance optimization this simplification appears to cause no problem.

3. COURSE STRUCTURE AND CONTENTS

We use Java for all programs in this class, to meet our goal to use a single programming language for the course. While the use of Java for threads may not require further discussion, the use of Java for message passing and OpenMP deserves a few comments. JCSP is a Java implementation of CSP (and more) that has also been used by other courses (e.g.,[12]). Our experience is mixed: although the implementation is fine, JCSP provides more than we need in this course. JOMP is a preprocessor-based OpenMP variant for Java. The current release does not support OpenMP 3.0 and includes no support for nested parallelism. As the development of JOMP seems to have stopped we are implementing a performance-competitive version. Neither JCSP nor JOMP are performance-sensitive implementations, but this is not a problem for our course (that pushes performance optimization into later courses).

We considered C++ or C# as alternative languages. But only Java allowed us to consider all three programming mod-

els in the context of a single programming language (we did not want to ask students to deal with the details of more than one new language in this course). Almost 3 years have passed since the original design of this course was done, and there has been considerable development regarding C++ and C#. Now that we have gained experience with Java, a future edition of this course may be based on C# or C++.

3.1 Topics

The class format is two lectures of 90 min each for 14 weeks. Table 1 lists the topics for each week. To increase cohesion, we use a few simple application kernels as running examples – these are revisited for each programming model – and include a simple simulation example (Game-of-Life) to show how parallelism is used in a larger context.

The two application kernels are mergesort and (dense) matrix multiplication. Both kernels are simple and easy to understand yet at the same time allow a discussion of various tradeoffs. In the assignments we do not ask the students to optimize their programs, so useful constructs like thread pools are not given much time. Some students pursue these directions anyhow, and we then discuss these approaches in the recitation groups led by the teaching assistants. However, as this material is not part of the syllabus, only a subset of the students may hear about these topics.

The hardest part in discussing threads and programming with a shared-memory data space is dealing with the visibility of updates. Since we do not want to discuss the complete memory model we focus on the use of “synchronized” methods. These methods establish global visibility of previous updates and allow the students to concentrate on the correctness of their programs.

The Game-of-Life example illustrates that in many programs the “parallel part” may only be a small part of the complete program. (Depending on the metric and the implementation, about 10% of the program is devoted to dealing with parallelism.) Some students complained about this fact and felt that material “unrelated” to parallel programming should not be covered or included in a class on parallel programming². However, the majority of the students liked the (trivial) example. The implementations of mergesort and dense matrix multiplication produce unimpressive output, so a problem that shows even simple output is appreciated (other institutions have successfully based their courses on parallel programming on graphics and animations[3]). The Game-of-Life assignment provides a bridge to the event driven programming practiced in many user interfaces and graphics projects and is included also for this reason.

4. DISCUSSION

4.1 Evaluation

We solicited feedback from the students of two years (2009, 2010). The survey was done anonymously. The students of the first edition (2009) were contacted two semesters after taking the course (to find out if any of the topics had indeed

²Anecdotal evidence suggests that the main reason for this complaint is that these students felt that they had to spend significant time on issues that (most likely) would *not* be part of the all-important first-year exam, which decides on their future as Computer Science students.

been revisited by other classes, as was hoped for by the curriculum design committee), students of the second edition (2010) were contacted about 3 months after the course had ended. About 45% of the 2009 students responded; for the 2010 students, the nominal return rate was higher (70%).

A response rate of 45% might appear low but ETH employs a model for student selection that is different from the model used by North American institutions. Enrollment in the 1st year is open to all students with an appropriate high school diploma; an exam at the end of the first year serves as a “delayed entrance exam” to decide who is allowed to continue. In Computer Science (at ETH), approximate half of the first year students do not pass this exam. So a response of 45% after one year means that we obtained a response from about 70% of those students that are still at ETH one year after the class (either because they passed the first-year exam or because they still have a chance to pass this exam). As the 2010 survey was taken prior to the first-year exam it is impossible to know the success of the respondents on the first-year exam.

Overall, the success rate for this course is about the same as the success rates for the other exams that make up the first-year exam.

4.2 Recurring themes

56% of the students of the first edition of this course reported that at least one of the topics from the introduction to parallel programming had reappeared in a later class. (31% were not sure, only 10% stated that no topic had reappeared). The topics that were listed as recurring are

- Locking (event queues, read-write locks, fairness)
- Semaphores
- Threads, shared memory programming, synchronization
- OpenMP

These topics resurfaced in several subsequent classes: operating systems, networks, system programming, computer systems lab, databases, advanced classes on parallel programming, software architecture, networks. Given that there were only two semesters that followed the 2009 edition of the class, we are happy with this result; as the students complete their B.S. program, we expect to add additional courses and topics to these lists.

4.3 Evolution

When asked what topics should be emphasized or de-emphasized in future editions, the responses (absolute numbers) were as follows. Of the 2010 population, 9 students make suggestions for additions, 20 students make suggestions to drop/de-emphasize material. The corresponding numbers for 2009 are 7 and 10. A number of topics find supporters and detractors.

The topic with the greatest number of requests to drop or de-emphasize are Java-specific issues. More than half of the 2010 students who responded to this question suggested that Java-related issues be dropped (to put this in perspective, less than 20% of the 2009 students made this suggestion). Of course, it is important to recall that in both populations, less than 25% of the students responded to this question.

Week	Lecture	Lab/assignment
1	Intro, parallel Algorithms	Warm-up
2	Java issues	Java practice
3	Locks	Bounded buffers for multiple threads
4	Condition queues	Mergesort
5	Visibility	Dense matrix multiplication
6	Safety and liveness	
7	Semaphores	Read-Write locks
8	Message passing	Dining philosophers
9	CSP	Mergesort
10	Programming with channels	Game-of-Life
11	Problem decomposition for parallel computing	
12	OpenMP	Proving program properties
13	OpenMP	Dense matrix multiplication
14	Models	Proving program properties

Table 1: Summary of course schedule.

Topic	Decrease Coverage	Increase Coverage
Message passing	2	0
OpenMP	4	3
Locks	5	2
Java/graphics	12	1
Java.util.concurrent		4
Reasoning about programs	5	1
Other		3

Table 2: Requests to change course content.

The use of Java deserves a bit of further discussion. In 2010, 36% of the student self-assessed their knowledge of Java to be “good enough to write a Java program of the scope/complexity of Game-of-Life” or better at the start of the semester. In 2009, the corresponding percentage is 31%. This assessment is a little surprising as the 1st semester uses a different programming language (Eiffel, an object-oriented programming language with static typing). In both years, about a quarter (2009: 27%, 2010: 23%) have no prior knowledge of Java, and for this reason, we include at the start of the semester extra lectures and labs to ease the path of these students. Not surprisingly, the reduction of the treatment of “Java issues” is requested by the segment of the student population that has already prior Java experience. Our response is that in future editions of this class we will form recitation groups based on the students’ prior knowledge of the implementation language and add more lectures outside of the course to assist those without prior Java knowledge.

Table 2 summarizes the results. Several students requested other languages, with C++ and Erlang receiving the most support.

4.4 Other issues

The course provides breadth but does not produce students that can immediately solve large-scale real-world parallel programming problems. The omission of performance optimization is one serious shortcoming.

The focus on low-level aspects has also the positive side-effect of improving the student’s programming skills. After a first semester that emphasized an object-oriented intro-

duction to programming, later classes (e.g., on operating systems, compilers, computational science) desired that students have a better understanding of program construction. So the introduction to parallel programming in the same semester as the class on “Data Structures and Algorithms” (the local instantiation of CS-2) serves as another vehicle to teach students the constructive aspects of programming. By exploring parallel implementations of mergesort after sorting has been covered earlier in the semester in CS-2, this topic is reinforced and the students investigate another aspect of sorting.

It is too early to evaluate the impact of this class on other classes in later stages of the curriculum. Several classes (on networking and on object-oriented programming concepts) removed material covered in this class. One instructor noted that his course is built on the assumption that students are familiar with parallelism (resp. concurrency) and anecdotal evidence suggest that this assumption is met. The use of Java has been received warmly; one instructor noted that in the networking class, there is a significant difference between “before” (the addition of this course to the curriculum) and now. For others (e.g., software engineering project) no difference was noted or there is no issue (as those classes use C or C++).

The absence of a single textbook that addresses this level of parallel programming is an issue. We recommend (and draw material from) the books by Herlihy and Shavit[11], Goetz et. al.[9], and Ben-Ari[2]. The later chapters of the first two books provide material for later classes as well. Nevertheless, the students seem reluctant to use books as studying tools. Only about 20% of the students admit in the evaluation that they use a book (the above three books receive the largest numbers of mentions). Anecdotal evidence from discussions with other instructors, at ETH and elsewhere, suggests that other classes have the same experience. Even if a course is based on a well-designed textbook (by authors who were recognized for their contribution to Computer Science teaching), the percentage of students that actually reads the book is also about 20%.

15 years ago, teaching a class on parallel programming required a substantial investment in computing infrastructure (see, e.g., Schaller et al. for a discussion[18]). The situation has changed a lot – students now own parallel computers. But whereas Schaller et al. anticipated that separate courses

on parallel programming would disappear (because the subject will be included in many places in the curriculum), there has been such a growth in concepts and techniques that we think it may be a while before these authors' prediction is realized.

5. CONCLUSIONS

Teaching parallel computing to undergraduates in their first year is a challenge. But the benefits outweigh the disadvantages. By exposing all students to the problems early on we capture the student's attention while they are still entering the field. A benefit is that students know about parallelism as an option and when future classes come along, they are aware of parallelism.

By discussing different parallel programming models we prepare the student for later. The student has a higher-level view than can be obtained if a class covers immediately specific programming tools. When the student encounters MPI later on, MPI is seen as a specific implementation of a general programming style. (And as students have understood the advantages and disadvantages of message passing, they can appreciate the many special functions included in MPI to allow a programmer to optimize performance.)

One effect of placing the course early is that we excite students. There is a small gap between the course content and leading edge developments, and students recognize that the topics that are covered in this course are hot. No other discipline can provide 1st-year students a similar opportunity. Students in physics or civil engineering must master a much larger body of prerequisites before they can be exposed to the leading edge. The Computer Science students that participate in such a class have a front-row view of an exciting evolving part of the field.

Acknowledgments

I thank the reviewers for detailed feedback and suggestions. I appreciate the comments by Susanne Cech Previtali and Christoph von Praun on earlier drafts of this paper. I benefited from discussions at the Computing Frontiers 2010 workshop on "Teaching Parallel Programming" organized by Stephanie Brumat. I am grateful to my colleagues Gustavo Alonso, Peter Mueller, Timothy Roscoe, and Peter Widmayer who provided feedback on the course and its relationship to other classes in our program. Finally, I thank the teaching assistants and students of "0024 - Introduction to Parallel Programming" (Spring 2009 and 2010) for their comments and patience. The development of the course was supported, in part, by a gift from Intel Corp.

6. REFERENCES

- [1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking parallel languages and Hardware. *Commun. ACM*, 53(8):90–101, 2010.
- [2] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd ed)*. Addison Wesley, Harlow, UK, 2006.
- [3] K. B. Bruce, A. Danyluk, and T. Murtagh. Introducing Concurrency in CS 1. In *SIGCSE '10: Proc. 41st ACM Technical Symp. Computer Science Education*, pages 224–228, New York, NY, USA, 2010. ACM.
- [4] D. P. Bunde. A Short Unit to Introduce Multi-Threaded Programming. *J. Comput. Small Coll.*, 25(1):9–20, 2009.
- [5] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge, MA, 2008.
- [6] Nvidia Corp. *Nvidia CUDA (C Programming Guide Version 3.1.1)*. Santa Clara, 2006–2010.
- [7] D. J. Ernst and D. E. Stevenson. Concurrent CS: Preparing Students for a Multicore World. In *ITiCSE '08: Proc. 13th Annual Conf. Innovation and Technology in Computer Science Education*, pages 230–234, New York, NY, USA, 2008. ACM.
- [8] A. L. Fisher and T. Gross. Teaching Empirical Performance Evaluation of Parallel Programs. In *SIGCSE '92: Proc. 23rd SIGCSE Technical Symp. Computer Science Education*, pages 309–313, Kansas City, MO, March 1992.
- [9] B. Goetz with T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley, 2006.
- [10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd ed)*. MIT Press, Cambridge, MA, 1999.
- [11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, 2008.
- [12] D. Jackson. A Mini-Course on Concurrency. In *SIGCSE '91: Proc. 22nd SIGCSE Technical Symp. Computer Science Education*, pages 92–96, New York, NY, USA, 1991. ACM.
- [13] M. C. Jadud, J. Simpson, and C. L. Jacobsen. Patterns for Programming in Parallel, Pedagogically. In *SIGCSE '08: Proc. 39th SIGCSE Technical Symp. Computer Science Education*, pages 231–235, New York, NY, USA, 2008. ACM.
- [14] D. A. Joiner, P. Gray, T. Murphy, and C. Peck. Teaching Parallel Computing to Science Faculty: Best Practices and Common Pitfalls. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pages 239–246, New York, NY, USA, 2006. ACM.
- [15] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.
- [16] S. Rivoire. A Breadth-First Course in Multicore and Manycore Programming. In *SIGCSE '10: Proc. 41st SIGCSE Technical Symp. Computer Science Education*, pages 214–218, New York, NY, USA, 2010. ACM.
- [17] N. C. Schaller. Panel: Parallel Computing in the Undergraduate Computer Science Curriculum. In *Supercomputing '91: Proc. 1991 ACM/IEEE Conf. on Supercomputing*, page 148, New York, NY, USA, 1991. ACM.
- [18] N. C. Schaller and A. T. Kitchen. Experiences in Teaching Parallel Computing—Five Years Later. *SIGCSE Bull.*, 27(3):15–20, 1995.
- [19] G. Wolffe and C. Treftz. Teaching Parallel Computing: New Possibilities. *J. Comput. Small Coll.*, 25(1):21–28, 2009.