A Breadth-First Course in Multicore and Manycore Programming

Suzanne Rivoire
Sonoma State University
Department of Computer Science
Rohnert Park, CA, USA
suzanne.rivoire@sonoma.edu

ABSTRACT

The technique of scaling hardware performance through increasing the number of cores on a chip requires programmers to learn to write parallel code that can exploit this hardware. In order to expose students to a variety of multicore programming models, our university offered a breadth-first introduction to multicore and manycore programming for upper-level undergraduates. Our students gained programming experience with three different parallel programming models, two of which are less than five years old and targeted specifically to multicore and manycore computing. Assessments throughout the semester showed that the course gave students a broad base of experience from which they will be able to understand ongoing developments in the field.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Human Factors, Languages

Keywords

parallel programming education, multicore, OpenMP, TBB, CUDA

1. INTRODUCTION

Over the past five years, processor manufacturers have abruptly shifted their method of scaling performance from increasing clock speeds to increasing the number of cores on a chip. Two- and four-core processors are common now, and the number of cores is expected to scale into the manycore range of tens and even hundreds in the coming years [2]. If these hardware improvements are to translate into software performance gains, programmers must learn how to write parallel code.

Researchers have proposed many programming models for harnessing multicore and manycore processors. However, despite increasing consensus that undergraduates should learn parallel programming, these newer models are generally presented only at the graduate level. Furthermore, it is unclear which of these models will prevail in the long term.

At Sonoma State University, a mostly undergraduate U.S. public university with approximately 125 computer science majors, we created an elective course to give our undergraduates a broad-based background in multicore programming. We wanted the students to gain plenty of programming experience but to avoid committing to a single model of parallel programming. Instead, our course aimed to give students the broad perspective necessary to understand future developments in this rapidly changing field and to effectively use whichever programming models win out. In order to meet this goal, we offered a unique breadth-first introduction to multicore and manycore programming, giving students hands-on experience with three programming models: OpenMP, TBB, and CUDA. The unusual organization and content of this course allowed our students to achieve these learning objectives, and the students' feedback has provided us with clear indications of what worked and what decisions should be revisited in future offerings of the course.

This paper first reviews related work in parallel programming education. Section 3 then discusses the content and structure of the course. Section 4 examines student feedback on the course organization and content, and Section 5 discusses lessons learned.

2. RELATED WORK

While the multicore era has introduced new urgency and complexity to parallel programming education, undergraduate parallel programming courses are not new [3, 8]. One approach to teaching undergraduates parallel programming is to inject it in snippets throughout the CS curriculum, as the University of Wisconsin at Eau Claire has done [7]. Another way to introduce new parallel programming models is in the operating systems class, where undergraduates traditionally encounter concurrency for the first time, as UT-Austin has done with transactional memory [19]. This paper presents an alternative approach: a single undergraduate elective course devoted to studying a range of parallel programming models. The advantages of this approach are a sustained focus on parallel programming, exposure to a variety of programming models, and ease of implementing the course without requiring systemic changes to the curriculum.

Beyond the undergraduate level, several studies have considered best practices for parallel programming instruction [13, 14]. At the graduate level, a course at the University of Central Florida examined several graphics processors and programming models [9]. The course described in this paper examines a broader variety of programming models with less emphasis on the hardware implementation. Because our course targets a less experienced undergraduate audience, the pedagogical techniques and assessments also differ.

A related body of research tracks the productivity of novice programmers with different parallel programming models [10, 11]. These studies target the high-performance computing domain, a relatively specialized area. However, as multicore processors become mainstream, the usability of parallel programming models will become even more important.

3. COURSE DESIGN

CS 385, Multicore and Manycore Programming [18], was offered as a special-topics upper-division elective for computer science majors at Sonoma State University. The course prerequisites were CS2 and introductory computer organization. Eighteen students, all junior- and senior-level undergraduates, enrolled. The class met for a total of two hours of lecture and two hours of supervised lab each week.

3.1 Subjects covered

The course content comprised five modules of two to three weeks each. The first module was a lecture-based crash course in parallelism, the memory hierarchy, and performance analysis. Because junior-level computer architecture was not a prerequisite for this course, an introduction to these topics was needed to ensure that all of the students had sufficient background.

The next three modules covered three different parallel programming models, which were chosen on the basis of their accessibility to our students (who are most proficient with C++), the robustness and user-friendliness of their implementations, and their potential scalability to tens if not hundreds of cores. Students completed labs and programming projects using each of these three models.

The first model covered was OpenMP [4]; the idea was to get the students up to speed using a simple and widely supported shared-memory programming model. OpenMP allowed the students to quickly express task- and data-level parallelism as well as some ability to tune the performance of their code through scheduling directives.

The next model was Intel's Threading Building Blocks (TBB) [12], which supports parallelism using STL-like C++ generic programming abstractions. TBB supports a greater variety of parallel patterns than OpenMP, and it largely takes low-level scheduling out of the programmer's hands.

The final programming model was nVidia's CUDA [16], a model for general-purpose programming of graphics processors (GPUs) that has been widely used to harness their ample data parallelism for scientific computing. Unlike current CPUs, GPUs are already able to handle thousands of hardware threads, which makes the mapping of tasks to threads substantially different than on CPUs. GPUs also require more explicit memory management on the part of the programmer, and all but the simplest tasks require some low-level hardware knowledge.

The remaining portion of the course was a qualitative overview of other parallel and multicore programming models. Students read and discussed technical papers on GPU computing in general [17], MapReduce [6], and transactional memory [1]. The students thus gained a wider perspective than would be possible with programming alone, and they also were exposed to reading technical papers and graduate-style discussion for the first time.

3.2 Assignments

3.2.1 Labs

In each weekly lab session for the first four modules, students wrote parallel code to perform a specific task, and they were asked to measure the performance of their sequential and parallel code at different data set sizes in order to understand speedup and overheads.

The lab assignments given during the initial background module required the students to mimic some of the thought processes of parallel programming before they were introduced to a specific programming model. The students wrote straightforward sequential versions of two very simple algorithms: incrementing all of the elements in an array, and finding the maximum element of an array. Both algorithms are "embarrassingly parallel," although the second requires a reduction at the end. The students then modified their code to operate on a chunk of the array at a time based on a "thread ID" parameter passed into their function, which introduced them to the problem of mapping elements of the array to "threads" correctly and efficiently.

In the next module, when they learned OpenMP, they actually parallelized this code and experimented with the effect of different data set sizes and scheduling directives on speedup and performance. They also implemented a simple task-parallel program. The final OpenMP lab presented a task with significant interaction between threads. The problem was to parallelize a program that scans through a character array and does the following:

- If the i^{th} character is 'q' or 'Q', it encrypts character i and the next 15 characters using an algorithm similar to AES, then copies the result into the output array and resumes with character (i+*16*).
- Otherwise, it copies character i to the output array unmodified.

This seemingly simple problem requires significant interaction between threads to ensure that the parallel version yields the same result as the sequential version, due to the difficulty of determining whether a given 'q' will begin a 16-character block, or whether it will be passed over because it is contained in the block triggered by a previous 'q'.

The TBB lab assignments used the array increment, array max, and "encryption" problems as well as specific smaller problems to exercise the timing features and concurrent containers unique to TBB.

The CUDA labs focused on the problem of finding the Euclidean distance between two N-dimensional points. This problem has an embarrassingly parallel phase followed by a reduction, which the students optimized extensively, gaining intimate knowledge of the hardware.

The final "labs" were writeups of the technical papers the students read in the final module of the course. Following the "just-in-time" teaching method successfully employed by Davis in upper-division CS courses [5], I prepared a list of questions for the students to answer by electronic submission, due a few hours before class. I then designed the in-class discussion around their responses. The assignments also included guidelines on how to read these papers and what the students should be looking for, since they had not read detailed technical papers before.

3.2.2 *Projects*

In addition to the supervised labs, the students were also assigned four longer-term projects. Three were programming projects: one for each of the three programming models studied. In these projects, the students were asked to parallelize matrix multiplication using the chosen model and then further tune the performance with at least four optimizations. Students were also asked to measure the of each version of the program and analyze the results. Finally, open-ended qualitative comments about the project or programming model counted for 5% of the assignment grade.

The fourth project was not initially planned as part of the course. Rather, it occurred to me midway through the course as a way to diversify the projects and to give students practice in technical communication. Since students had found that the existing documentation and resources for some of the programming models failed to address novice parallel programmers, I assigned them to write tutorials about one of the three programming models studied. The students worked in groups of 2-3 students (8 groups total). Each group was assigned a programming model based on their stated preference and on a random drawing to be sure that each model was equally represented. This project was a valuable addition to the course and a useful assessment of the students' high-level understanding. However, because it was not initially written into the course schedule, the students ended up with an unnecessarily heavy workload when it was assigned toward the end of the semester.

3.2.3 *Other assessments*

The only other assessments in the course were 4 narrowly focused quizzes (one for each module except the introduction) and a comprehensive final exam requiring students to synthesize their knowledge of all of the different programming models studied.

4. STUDENT FEEDBACK

4.1 Course organization and content

I solicited qualitative and quantitative student feedback on the course organization and content in an informal survey at the end of the semester. I also examined the qualitative responses from our standardized institutional course evaluations; unfortunately, the quantitative portion of our institutional evaluations is focused solely on the instructor rather than the course. The data show an overall positive response to the course, although they indicate possible areas for improvement the next time this course is offered.

Figure 1 shows the results of the quantitative portion of this student survey, which was answered by 15 of the 18 students. Students were asked to rate the three major instructional components of the course—lectures, labs, and projects—on their helpfulness in four major categories: understanding parallel algorithms, analyzing the performance of code, understanding the latest developments in the technology industry, and increasing programming skills.

Option	**# students**
Would have preferred depth-first (studying one model in great detail)	0
Would have preferred depth-first with a little bit of breadth	7
Liked the course as is	7
Would have preferred a little more breadth (more topics, shallower coverage)	0
Would have preferred a lot more breadth	0
Don't care/don't know	1

Table 1: Student opinions of course breadth

The figure shows that the project portion of the course was most polarizing, with high variation in student ratings across all four of the questions. Projects also received the lowest average ratings of any course component in three of the four categories. Furthermore, in the category of "helped me understand and analyze the performance of my code," at which the projects were specifically targeted, they were rated as no more helpful than the lab assignments. A look at the students' qualitative remarks provides no easy answers; students commented both in favor of and against using the same problem for all of the different programming models, and only one student complained about the choice of matrix multiplication. One possible explanation is the students' understandable frustration with competing with other students for the limited hardware resources available to extensively test the performance of their code, which one student summed up as "a lesson in agonizing frustration."

The labs, on the other hand, were rated as the most helpful component of the course in three of the four categories shown, and they seem to have succeeded where the projects failed. Since the lab exercises presented the students with a variety of types of parallelism, it is not surprising that students rated them highly in the category of parallel algorithms. Similarly, the increase in programming skills may be attributable to the cooperative setting of the lab portion of the course, which allowed students to quickly get feedback from peers and the instructor.

Finally, the evaluations showed that students were highly enthusiastic about learning a topic that they perceived as relevant and cutting-edge. The lack of a textbook and the relative paucity of beginner-level tutorials about TBB and CUDA did not seem to bother the students; in fact, the only qualitative comment on this subject urged me to continue encouraging the students to find and read online documentation in lieu of a textbook in the future.

Perhaps most importantly, I solicited student feedback on the choice to make this course breadth-first instead of pursuing a single parallel programming model (see Table 1). Students' responses to this question divided evenly between "keep as is" and "would prefer depth-first with a little bit of breadth." In the qualitative comments, students mentioned that they appreciated being able to compare parallel programming models, but some students did suggest studying two models in detail instead of three.

4.2 Programming models studied

I also obtained student feedback on the three program-

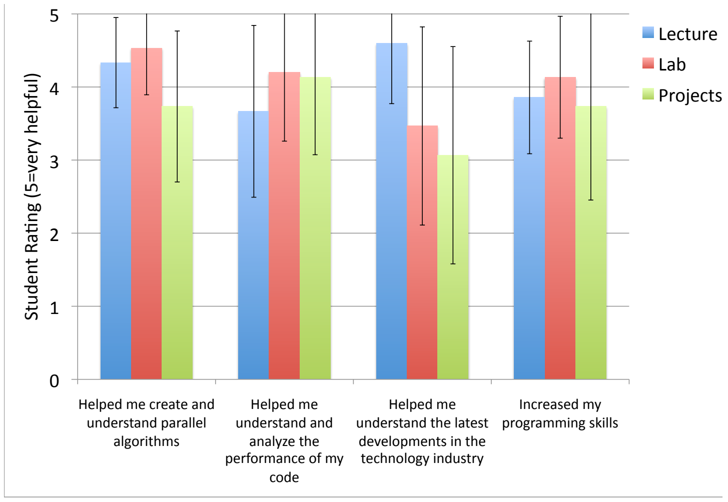

Figure 1: Results of end-of-semester student survey

	OpenMP	TBB	CUDA
1st choice	0	1	7
2nd choice	6	1	0
3rd choice	1	6	0

Table 2: Number of groups choosing each programming model for the final project

ming models we studied from their three programming project writeups and from their preference ordering of the three models for the tutorial project.

Table 2 shows the students' order of preference for tutorial topics. CUDA was the first choice of seven of the eight groups; the group that chose TBB did not specify a second or third choice because they were well aware that they had no competition for their first choice. OpenMP was the clear second choice, and TBB the third. The qualitative end-of-semester feedback, as well as student comments on project writeups, further underscored this order of preferences. These preferences also correlate with the degree of low-level detail the programmer is concerned with: CUDA requires the programmer to understand the hardware on which it runs at a fairly low level, while TBB nudges the programmer toward a higher level of abstraction and less concern with the low-level details of thread mapping and scheduling. Their preference for lower-level models, surprising given their prior experience with the STL and their lack of similar background in computer architecture, was summed up by one student in his CUDA writeup:

"My opinions on CUDA have changed since we first began working with it. At first, I didn't like it because it was for GPUs, and I'm not really interested in graphics... Then we started learning a bit more about GPUs, and I've realized that they are for so much more than graphics now... I began to like CUDA more and more, as I began to really understand how I am working with the hardware at such a low level. TBB was thinking in terms of high-level programming, object-oriented, but with CUDA, I really have full control of what's going on. I think that this class in general has prepared me for Architecture and Operating Systems so much, and working with CUDA is a perfect example of this. I've really come to understand CPU and GPU hardware, and I'm starting to like it."

5. LESSONS LEARNED

5.1 Breadth-first?

While I continue to believe that undergraduate students are better served by exposure to a variety of multicore programming models than a semester-long commitment to a single model, experience with this course shows that 3-week modules may be too short for students to get an adequate feel for a programming model. One possibility is to focus the programming projects on two models but introduce breadth through the readings, since the final phase of the course, in which we discussed papers on other programming models, went better than expected, . Another alternative, also suggested by some students, would be to require more programming and architecture background of the students, requiring less time for introductory material and allowing more time to study each model. Unfortunately, the latter alternative is probably impractical at our institution given its relatively small size; we are unlikely to have a sufficient number of students who meet a significantly more stringent prerequisite.

The choice of programming models remains an open question for future offerings of the course. I would definitely include a GPU-based programming model, whether CUDA, OpenCL [15], or something else; the high degree of parallelism gave students the satisfaction of seeing dramatic speedups, and the low-level control required pushed the students to understand hardware at a deeper level. The other model(s) are less obvious; using OpenMP as a gentle introduction to parallel programming turned out to probably be unnecessary, but TBB was roundly disliked and students found it difficult to understand the performance of their TBB programs. Since this area is evolving so quickly, perhaps the right platform has not yet been developed.

The paper discussions were another means of achieving breadth, and the specific papers chosen gave students a wider perspective on parallel programming at a level that they were able to understand. In the future, we would make up for decreasing breadth in the programming projects by budgeting time for a wider range of readings on parallel programming. This method would give the students the desired exposure to several programming models without the overhead of repeatedly mastering new syntax.

5.2 Improving the course components

Student feedback shows a clear opportunity to improve the projects. Most significantly, as important as it is for students to learn how to optimize code with the assistance of thorough performance measurements, contention for hardware resources can render the experience both meaningless and frustrating. Matrix multiplication is also a rather staid problem for the students to optimize. The right solution may therefore be to give students a more complicated but more interesting problem and emphasize getting a working implementation at the expense of optimizing and tuning.

The lab activities were mostly intended to get students comfortable with the syntax of the different programming models, but the data in Section 4 show that they were integral to achieving the conceptual learning objectives of the course. This indicates an opportunity to revisit the content of the labs to increase their effectiveness in that role.

Finally, the students' engagement with the readings far exceeded my expectations, largely due to the reading guides and the just-in-time teaching methods used for that component of the course. Due to the success with the reading guides in this class, I have begun to prepare reading guides for other, textbook-based, courses as well; after all, college textbooks may be as unfamiliar for beginning students as technical papers are for advanced students.

6. CONCLUSION

In conclusion, a breadth-first introduction to multicore programming models satisfied my objectives of giving students a broad-based perspective on parallel programming, and students indicated general satisfaction with the course. The organization of the course and the specific combination of models studied were novel and proved effective for our students, with some caveats.

The choice of a GPU programming model was pedagogically effective and popular with students; in fact, students enjoyed and learned more using low-level models than high-level ones. This counterintuitive observation will help me select the other programming models more effectively in the next course offering. The success of the reading portion of the course also suggests that we could cover two models in depth instead of three and compensate by reading about a wider range of models. While there is plenty of room for adjustment, this initial version of the course successfully prepared our undergraduate computer science students for productivity with emerging parallel programming models.

7. ACKNOWLEDGMENTS

I am grateful to the faculty of the Sonoma State University CS department for the opportunity to offer this course, to Christos Kozyrakis for his advice on the course content, and to Roger Mamer for his technical support throughout the semester. In addition, this paper benefited from the comments of John Davis, Joy Tang, Kathy Charmaz, Sheila Katz, Richard Senghas, Tom Rosin, and the anonymous reviewers. Finally, I am especially grateful to the students who took CS 385 in Spring 2009 for their enthusiastic participation in, and their patience with, this experimental course.

8. REFERENCES

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007.

[2] K. Asanovic, R. Bodik, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] R. M. Butler, R. E. Eggen, and S. R. Wallace. Introducing parallel processing at the undergraduate level. *SIGCSE Bull.*, 20(1):63–67, 1988.

[4] L. Dagum and R. Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1), 1998.

[5] J. Davis. Experiences with Just-in-Time Teaching in systems and design courses. In *SIGCSE*, 2009.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[7] D. J. Ernst and D. E. Stevenson. Concurrent CS: preparing students for a multicore world. In *ITiCSE*, 2008.

[8] A. L. Fisher and T. Gross. Teaching the programming of parallel computers. *SIGCSE Bull.*, 23(1):102–107, 1991.

[9] H. Gao, M. Dimitrov, et al. Experiencing various massively parallel architectures and programming models for data-intensive applications. In *Workshop on Computer Architecture Education*, 2008.

[10] L. Hochstein, V. R. Basili, et al. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008.

[11] L. Hochstein, J. Carver, et al. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing (SC)*, 2005.

[12] Intel. Intel Threading Building Blocks (TBB). `http://software.intel.com/en-us/intel-tbb/`.

[13] M. C. Jadud, J. Simpson, and C. L. Jacobsen. Patterns for programming in parallel, pedagogically. In *SIGCSE*, 2008.

[14] D. A. Joiner, P. Gray, et al. Teaching parallel computing to science faculty: best practices and common pitfalls. In *PPoPP*, 2006.

[15] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[16] nVidia. CUDA zone. `http://www.nvidia.com/object/cuda_home.html`.

[17] J. D. Owens, M. Houston, et al. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[18] S. Rivoire. CS 385: Multicore and manycore programming, Spring 2009. `http://rivoire.cs.sonoma.edu/cs385/`.

[19] C. Rossbach, O. Hofmann, and E. Witchel. Is transactional programming really easier? In *PPoPP*, 2010.