Third-Year Parallel Programming for CS Undergraduates

William B. Gardner
School of Computer Science, University of Guelph, Guelph, ON, Canada

Abstract - *This paper describes a successful new course aimed at helping soon-to-graduate students move into jobs using current tools for parallel programming, by acquiring the theoretical background needed to keep abreast with rapid industry developments and to evolve with them. It intentionally spans the range of multicore to cluster computing, based on the same underlying principles. All aspects of the course are described, including textbook, schedule, lab content, assignments, projects, and outcomes after two offerings.*

Keywords: Parallel programming, Undergraduate CS curriculum, Multicore, High-performance computing

"I help search for water on Mars." At the first meeting of our new Parallel Programming course, rolled out experimentally as a Special Topic offering in Fall of 2009, we were going around the room, each student invited to say why he or she had registered and what they hoped to learn. Among the expected third- and fourth-year CS undergraduates was this one graduate student from the Physics Department. Her research group analyzed spectrographic data radioed back from the Mars Rover for telltale signs that water molecules could be present. In order to interpret the spectrograms, they would compare them to those of simulated bombardments by 50 billion photons onto various artificial compositions of Martian "soil" spiked with specific amounts of water. Each run for a single soil/water composition would take 24 hours to complete on a high-end desktop computer, and she was the only person in the lab who considered this state of affairs decidedly subpar for the new millennium of computing!

For CS students who were used to coding assignments that instantly compiled upon pressing Enter, and executed in tens of seconds at most, this was a revelation. In the event, she lacked the programming background to stay with us, but her case study provided invaluable motivation for launching the course.

1 Background and introduction

Our School offers a Bachelor of Computing degree with two majors, classical computer science and software engineering. For the last several years, it was obvious that we would have to come to grips with teaching parallel programming beyond the basic introduction to concurrency that has long formed a topic in Operating Systems (OS). As the faculty member whose research has centered on formal methods for specifying concurrent systems [1], with background in digital hardware as well as software engineering, I was relatively suited to the task of examining our options and creating a suitable course.

Specifying the OS course as a prerequisite, the 2009 Special Topic attracted 17 students. It was repeated, again as a Special Topic, in Fall 2010, with modifications based on the first experience. This time 26 students signed up, including another graduate student in Physics (this one stayed and did an impressive project). Based on student surveys, the course can clearly be called a success. By Fall of 2011, it will enter the regular curriculum as CIS*3090 Parallel Programming, an annually offered elective.

This paper covers every important aspect of the course, for the benefit of those who may wish to copy or adapt it, including the following:

- Approach: why we chose to mount a single special course, as opposed to incorporating parallel programming topics into existing core courses
- Characteristics: breadth over depth, wide-spectrum (multicore to high-performance clusters), theory-based with programming practice, tools-oriented (for profiling and error detection), programming techniques selected for prospects of longevity, special-hardware focus avoided, assumed prerequisites, target audience
- Organization: lecture topics, 12-week schedule, programming platforms, software tools

- Textbook: strengths and weaknesses
- Assignments: rationale, coordinated hands-on lab sessions, contests
- Project component: categories, sample of topics, approach to evaluation
- Human resources: instructor workload, use of teaching assistants

2 Approach

From the standpoint of impact on curriculum, the first question to settle is whether a CS department will attempt to integrate parallel programming concepts into several existing core courses, or will mount a standalone course, or even do both. With an eye to the burgeoning multicore future, it seems wise to start "parallel consciousness raising" early, and we do not oppose that approach. Nonetheless, pursuing some degree of integration does not detract from the benefits of mounting a full course—in particular, training students to have job-ready skills—and indeed could allow that course to have a more advanced starting point. Furthermore, in a department such as ours, there are enough drawbacks and stumbling blocks so as to make it a choice between mounting a standalone course right away, versus doing nothing helpful for possibly a long time. In the end, we took the standalone approach for several reasons:

First, it was less disruptive to the faculty and curriculum as a whole. The core courses in our 12-week semesters are already "full," so that adding topics necessarily means displacing something else, which, if shifted around, can widen the impact to other courses.

Second, while it has been suggested that we should cease teaching sequential programming and just carry on with parallel programming [2], this is not convincing. Parallel programs are made of sequential portions, and applications that don't require special performance will continue to be written adequately as sequential codes. Similarly, going back to the early prediction that object-oriented programming (OOP) would displace procedural programming, this has not occurred, and we see that OO methods are made up of procedural code. Like many departments, we continue to teach both styles of programming with full courses, and there is no good reason to think that sequential programming courses can simply be converted to parallel courses. For a number of years to come, it will be necessary to take up the latter as additional subject matter.

Third, there is the problem of personnel. Given that most present faculty members were educated in the sequential programming era, how are they going to readily teach parallel topics? It is more straightforward to concentrate the responsibility for that new knowledge and skills in some volunteers who already have suitable background, and let them create a course.

Finally, model curricula with integrated parallel topics are only now being proposed (for example, see [3]), and being an early adopter is risky. Many departments will prefer to learn from others' experiences before tampering with their own core courses, which may be based on textbooks that do not have parallel topics. In contrast, the investment in a standalone course can pay benefits right away in terms of student satisfaction. And if the department eventually moves to introduce parallel topics early, the programming course can adapt by starting from a more advanced level, which is a win-win outcome.

3 Characteristics

What is the target audience of students, and which prerequisites should be demanded? We decided to aim for third- and fourth-year students, for whom programming per se is no longer a big challenge, thus they are free to concentrate on grasping new concepts and applying them with the programming skills they already possess. Two essential elements of background knowledge are concurrency—which in many OS courses is taught along with POSIX threads, and exposes students to critical sections, deadlocks, resource contention, mutexes, and condition variables—and basic computer architecture. For decades, computer and OS designers have dedicated themselves to making the hardware largely invisible to software, and may have succeeded too well. By now, we find that many students will only take an architecture course grudgingly, and feel there is no purpose in it. And yet, in learning how to get maximum speedups out of parallel programs, it is necessary to draw in issues like memory bandwidth limitations, cache coherency, and false sharing. If students are going to understand why a program's speed may depend on the arrangement of a data structure in memory, they have to know something about hardware. We find that our second-year Structure and Application of Microcomputers course gives enough exposure that they can follow the hardware issues related to parallel performance.

When parallel programming courses started to be mounted in universities, a typical approach was to focus on a certain architecture, e.g., IBM Cell BE or GPU. But, emphasizing special hardware carries risks of rapid platform evolution or obsolescence. We prefer to give students a broad introduction ranging from the now-ubiquitous multicore desktop to the previously established world of high-performance computing (HPC) clusters. These disparate platforms have both similarities and differences that help students understand the underlying hardware issues, and the advantages and limitations of various programming techniques. We chose to teach POSIX threads, OpenMP, and message-passing programming because they leverage, to some extent, concurrency background taught in the OS course, are widely practiced in industry and academia, and show no sign of going away.

4 Organization

Organizing any CS course is a major undertaking that necessitates consideration of textbooks or other learning resources, programming languages and computing platforms, often supplied via in-house computer labs, additional software packages and training in their use, plus plans for assignments, exams, and possibly term projects.

The various components of the course were weighted into the grade as follows:

- Assignments 30%
- Term project 35%
- Final exam 25%
- Participation 10%

The exam was based on the textbook and the three programming methodologies. Participation marks were given to encourage attendance at, and peer evaluation of, the project presentations.

Since the textbook is a kind of linchpin, the subsections below start with that, and then the schedule of lecture topics is presented, coordinated with textbook chapters. Next, the programming platforms and software packages are described. The plan for the programming assignments is given, and then for the projects. Finally, utilization of human resources is described.

4.1 Textbook

In 2008 and 2009, there were as yet few solid entries into the market for parallel programming textbooks. Fortunately, I have been very content to discover *Principles of Parallel Programming,* by Calvin Lin and Larry Snyder, Addison-Wesley, 2009. It strikes the right balance for a university setting, between mastering techniques and tools, on the one hand, and presenting a theoretical basis, on the other. The authors introduce their parallel pseudocode, called Peril-L, which is suitable for implementing as pthreads, OpenMP, or message-passing. Similarly, the theory component is equally applicable across the spectrum of parallel platforms. Sufficient hardware description is supplied to explain phenomena that must be grasped in order to produce scalable programs.

The first chapter is captivating: It commences straightaway with a simple case study that points out several common pitfalls in parallel programming, e.g., the parallel version runs slower than the serial version, race conditions produce incorrect results, it is not very scalable with more cores, and so on. From a student's standpoint, this immediately raises the stakes from "here we are, learning yet another programming language (which I could have taught myself)" to "maybe there is something I don't know after all!" This has the effect of strongly motivating the course, and it plays into an important theme: the *computer professional* knows how to obtain good results (here, parallel performance) by applying knowledge and skills; the *hacker* gets good results, if at all, mainly by luck.

The main weakness of the first edition—possibly a symptom of being rushed into a hot market—is a large number of errata, most of which are noted on the authors' website [4]. One can only hope that a second edition will be printed to solve these problems.

Two other books were put on the course syllabus as recommended reading: *Patterns for Parallel Programming,* by Mattson, Sanders, and Massingill, Addison-Wesley, 2005; and *The Art of Multiprocessor Programming,* by Herlihy and Shavit, Morgan Kaufmann, 2008.

The Lin and Snyder textbook is organized into sections. We studied the entire first section's five chapters, which provide the necessary conceptual basis for program development. Specific programming methodologies are covered in chapters 6 to 8, classified as threads, "local view" languages, and "global view" languages, respectively. One is free to pick and choose among them. Chapter 9 gives an assessment of existing approaches, and becomes rather abstract for our purposes. Chapter 10 surveys "future directions" and goes well alongside overviews of selected platforms I chose to introduce: NVDIA GPU with the CUDA language, OO threading libraries from .NET and Intel, and the IBM Cell BE. Anticipating use in a project context, the book ends with a practical "capstone project" chapter (11).

The overall timing strategy for the 12-week course involved three stages:

1. Laying the conceptual groundwork for parallel programming, based on chapters 1-5.

2. Learning three specific programming methodologies applicable to non-shared memory (cluster) and shared memory (multicore) platforms, with one assignment each.

3. Surveying a variety of topics while students were carrying out projects utilizing the above methodologies. If experts on specialized topics are available, this is an ideal time to bring in guest lecturers.

The schedule of topics, coordinated with textbook chapters and labs, is shown in Table 1. There was a definite purpose in the order of teaching the programming methodologies:

Table 1. Course schedule

Unit	Reading	Lab
1. Introduction	chs. 1 & 11 Getting Started	
2. Understanding Parallel Computers	ch. 2	
3. Reasoning about Performance	ch. 3 to Trade-Offs	Pilot library
4. First Steps Toward Parallel Programming	ch. 4	
5. Scalable Algorithmic Techniques	ch. 5	Intel Parallel Studio
6. Programming with Threads • POSIX Threads • OpenMP	ch. 6 (parts)	
7. Preparing for Project	ch. 11 Capstone	
8. Assessing the State of the Art	chs. 9, 3 & 11 (finish)	
9. Future Directions in Parallel Programming	ch. 10	

The first one, message-passing programming for high-performance clusters, utilizes an in-house library, Pilot [5], which is a simple process/channel abstraction layered on top of conventional MPI, and targeted at novice scientific programmers. Pilot's process definitions are similar to pthread_create (which students know from the OS course), and its distinctive C API is modeled on stdio's well-known fprintf/fscanf, so it is simple to teach and more difficult to abuse than MPI. Pilot also includes an integrated deadlock checker that is capable of diagnosing right to the line number in program code that misused of the API or caused a circular wait, for example, thus preventing the phenomenon of silent, hung programs commonly experienced by beginning MPI users.

The above features make Pilot very suitable to teach while the course content is just starting to build, and students can use it for the first assignment. Those who wish to do a cluster-based project on hundreds of processors can still use Pilot (which also has a Fortran API), or they can branch out and learn the more complex, low-level MPI, for which Pilot will have given them good preparation. Pilot is available for free downloading from its website [6], and installs with any MPI.

Next comes pthreads with Intel Parallel Studio tool support; and OpenMP, also with Parallel Studio. Pthreads is taught before OpenMP, both to connect back to students' OS course experience, and because OpenMP is deceptively easy to use, to the point of making them feel that Pthreads programming is too onerous. Assignments #2 and #3 involve implementing the same program in pthreads and OpenMP, which helps them to closely compare these technologies. The purpose of using Intel Parallel Studio (available under free academic license) is to employ its tools Parallel Inspector, for detecting shared memory conflicts and potential race conditions, and Parallel Amplifier, for profiling program performance down to the core level. These tools strongly support the common use case of parallelizing existing sequential programs. Students find it very illuminating to see just how much of the time their program spends utilizing *n* cores (where *n*=1-16 on our system).

4.2 Parallel programming platforms

In order to offer an exciting suite of high-end parallel hardware for assignments and projects, we were able to procure a "pre-owned" 32-node Itanium cluster from SHARCNET—our university's associated high-performance computing consortium—running Linux, and a 16-thread Mac Pro Core i7 running Windows Server under Apple's Bootcamp. (SHARCNET accounts were available for projects wanting hundreds of processors.) This enabled students to obtain hands-on experience typical of both the HPC world and systems used in industry. The same Intel C/C++ compiler was installed on both systems.

Intel offers a free Summer School with training on Parallel Studio. It is that training material that we use successfully for the course labs. The Pilot hands-on labs are the same half-day tutorial that we run at international conferences. Those labs are also available for downloading from the Pilot website.

4.3 Assignments

The students carry out one assignment using each of the three programming techniques, starting with Pilot. As stated above, the pthreads assignment gets rewritten using OpenMP for the purpose of comparing and contrasting.

A key requirement for all assignments is short written reports describing the rationale for the student's parallel design, and featuring timing, speedup, and efficiency graphs against an X axis of number of processors, all with the student's interpretations. Timing is done both with and without compiler optimization. Students are often shocked to see that the compiler can cut execution time by as much as one-third. They must provide proof of program correctness, and explanations for what they learned from refining their programs to improve performance. These write-ups showed that they were able to understand and apply the theory we had learned.

The three graphs basically show the same data in different ways, yet they are not redundant and one can learn something more from each of them. The timing graph (wall clock time) gets right to the obvious objective: how fast is my program? The speedup graph, showing the ratio of serial time vs. parallel time, serves to factor out compiler optimization effects, and yields a frank assessment of scalability or lack thereof. The efficiency graph, speedup divided by number of processors, shows how far their parallel performance falls short of the perfect "1.0" efficiency line.

Bonus points were awarded for the fastest solutions, in keeping with the parallel performance emphasis. Those finishing in second and third place were allowed to challenge the winner to a rematch, which gave them the experience of consciously trying to tune their programs, and often the ranking—and the bonus points—changed hands.

4.4 Projects

As important as programming assignments are, they have to be tightly specified and of limited scope. To really apply what they are learning, doing an independent project is invaluable. Students were allowed to work as individuals or form teams of two.

Each student or team has to propose (and get approved), program, and present to the class a project chosen from five categories, some of which come from the textbook. A written project report is handed in along with their source code. The categories, with samples from the two years, are as follows:

1. (Re)implement existing parallel algorithms: checker playing, cryptology, Quine-McCluskey method

2. Compete with standard benchmarks (no one chose this)

3. Develop new parallel computations: force histograms for 3D vector images, traffic simulator, gaussian blur

4. Pilot-related development: porting Pilot to Lua (LuaPilot) and Python (Pylot)

5. Exploring beyond the course, which could involve parallel languages that we did not study as a class: F# (solving Minesweeper), C# TPL (tree search) Cilk++, CUDA (particle simulation), OpenCL (image processing)

The approval step is advisable for ensuring that students do not recklessly launch into unsuitable topics, and that the scope is compatible with the few weeks left in the course, or that they at least have a fall-back strategy in case things go more slowly than they expect. There was no formal deadline for the proposal, which was intended to facilitate rapid turn-around by avoiding having them all submitted at once. Counter-balancing that flexibility, students were required to provide weekly progress logs of their project activities—they could read each other's logs—and points were deducted if the instructor's weekly spot check found no update. This served to keep them moving along.

This "exploring" feature is an excellent way to keep the course up to date, by encouraging students to try out the very latest technologies and report to the class, which in turn enables the instructor to freshen the course without having to continually revise the instructional components.

Because of the knowledge they have gained, it is typically straightforward for the students to relate most tools or techniques to concepts they have learned, comparing and contrasting with methodologies they all (now) understand and have experience with. This gives the students a lot of confidence when they see they are capable of understanding and trying out some new parallel programming technology on their own.

4.5 Human resources

In terms of instructor workload, the projects require the greatest time commitment, depending on how thorough one wishes to make the evaluation. I found the projects to be so interesting, that I wanted to build and run each one. This is probably the least scalable component of the course as it is currently constructed.

If teaching assistants are available, one good place to use them is for "hand-holding" in the lab sessions, that is, walking around and assisting anyone who is having problems doing the exercises. Another worthwhile use is for compiling and running all the submitted assignments, checking for correct output, and making timings for the contests. This leaves the instructor to read and evaluate the paper reports. Someone who took the course before makes an ideal TA, otherwise it would be difficult to find anyone qualified who has only come through standard CS training.

5 Conclusion

Today's computer science students are entering a new era in parallel computing, featuring cheap multicores and high-performance clusters, but have received traditional largely-sequential training. Based on our experience with this course, we found that resources are presently available to mount a standalone course relatively cheaply. It meets CS students' practical educational needs, and it can be extremely gratifying to teach.

One measure of success is how the students view themselves as parallel programmers. After each course, they were asked to fill in an anonymous survey including the statement "I think I can handle parallel programming" (0-10 scale), comparing before vs. after the course. Their confidence rose impressively, on average, from 2.8 to 9.2. That confidence, plus all the languages and tools to write on their resumes, should give them a significant employment advantage.

If, in the future, our department decides to begin integrating parallel programming topics into the existing core courses, that will simply strengthen students' preparation for this dedicated course.

6 References

[1] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.

[2] Wen-Mei Hwu, David Kirk, Christoph Lameter, Charlie Peck, and Michael Wrinn. There is no more sequential programming. Why are we still teaching it? In *Super Computing (SC08)*, Education Program, Panel Discussion, Austin, TX, Nov. 17 2008.

[3] NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: Core topics for undergraduates [online]. Available from: http://www.cs.gsu.edu/ tcpp/curriculum/index.php.

[4] Errata for Principles of Parallel Programming [online]. Available from: http://www.cs.utexas.edu/ lin/errata.html.

[5] John Carter, W.B. Gardner, and G. Grewal. The Pilot approach to cluster programming in C. In *Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium, Workshops and Phd Forum*, Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10), pages 1–8, Atlanta, Apr. 23 2010.

[6] Pilot home [online]. Available from: http://carmel.socs.uoguelph.ca/pilot.