Teaching concurrency beyond HPC

David P. Bunde [1], Jens Mache [2]

[1] *Department of Computer Science*
Knox College
Galesburg, IL USA
dbunde@knox.edu

[2] *Department of Mathematical Sciences*
Lewis & Clark College
Portland, OR USA
jmache@lclark.edu

***Abstract*—We present our view that teaching concurrent programming to a broad audience will require adapting materials developed by the High-Performance Computing (HPC) community to the different goals and interests of students outside this community. Specifically, in many applications achieving peak performance is less important than in HPC applications. Instead, students can be taught to focus on scalability and rapid implementation, only aggressively optimizing their programs if initial performance is unacceptable. In addition, new examples must be developed to teach concurrent programming, deemphasizing computational science in favor of applications that students find more intrinsically motivating. We list some of these examples and look forward to hearing additional ideas from others.**

Until recently, programmers got a "free lunch" in the sense that they could expect clock speed to increase in each processor generation, speeding up their programs without any additional effort on the programmer's part. With the topping out of processor speeds, increasing performance will depend on exploiting higher core counts. This will require many more programmers to write concurrent applications. Doing so was previously considered a specialized skill, one primarily practiced in the High-Performance Computing (HPC) community. A great challenge of our time is to teach a much broader group of computer scientists this skill so they can expose the concurrency of their applications and exploit multicore hardware. It is our contention that, although the HPC community has a lot to offer in meeting this challenge, we cannot simply teach all computer scientists an existing high-performance computing curriculum. Rather, we believe that teaching concurrent programming to the broader community of computer science students requires a shift away from the traditional focus on absolute peak performance, focusing instead on students' ability to write correct programs that scale, and the development of teaching materials with examples that are more appealing to students.

We begin by addressing the issue of peak performance. This is a long-standing goal in the HPC community, which strives to minimize the "time to science" for applications. Extraordinary effort is made to achieve this goal, with applications being carefully tuned using detailed knowledge of the hardware. The pursuit of higher performance leads to the construction of large systems, epitomized by those on the Top 500 list, as well as the exploitation of special types of hardware, such as the Cell processor and GPUs. All of this effort is appropriate for applications requiring absolute peak performance.

In contrast to this, the aggressive pursuit of performance is not central to most courses in a typical CS curriculum. Instead, the courses that explicitly compare ideas based on performance, data structures and algorithms, do so at a high level and typically ignore the implementation details so crucial to HPC-level performance. In typical programming courses, correctness is almost always the main goal, with poor performance only penalized in the most egregious cases. This emphasis on correctness occurs even in courses such as computer organization and operating systems whose material is crucial for achieving high performance, because it is hard enough for beginners to write correct programs. Once students can reliably write correct programs, they generally begin taking electives such as AI and graphics, where the focus is again on achieving a desired result rather than on performance per se.

We believe that this emphasis on correctness and capability over performance is appropriate even after the adoption of manycore hardware. This is not to say that performance is completely unimportant; multicore hardware has no reason for existing except to surpass serial performance. It is just that most programmers and most applications do not require peak performance. Consider the current practice of most sequential programmers: get a reasonable design working and then optimize it only if better performance is needed. This strategy places a high value on programmer time, avoids unnecessary optimization, and works well in a marketplace that often rewards being first to market. Analogously, we expect that an effective strategy for the multicore world is to write an initial program, focusing on correctness and scalability, and then further optimizing performance only if necessary. By including scalability in the initial design, the programmer ensures that the code benefits from successive hardware generations, just as in the era of increasing clock speed.

Since we do not advocate a focus on peak performance, we believe that it is appropriate to teach with high-level languages or frameworks that allow the programmer to express parallelism even if they hide some details and impede low-level optimization. This situation is analogous to the way

that many introductory courses currently use Java, which provides automatic memory management, array bounds checking, and extensive libraries, but at the cost of runtime overhead. Many potentially-suitable parallel languages and frameworks have been developed to facilitate concurrent programming; interesting examples include NESL [1], Fortress [2], Hadoop/MapReduce [3], Charm++ [4], and Cilk++ [5].

Whatever programming environment is used, the goal of core computer science courses should be to teach students key principles and techniques of concurrent programming such as mutual exclusion, load balancing, grain size, parallel prefix, and divide and conquer. These ideas will apply to any concurrent environment the students find themselves using in the future and form a foundation of "algorithmic taste", the intuition our students will use to select initial designs with the desired scalability properties. In addition, we must teach our students how to analyze and improve their algorithm or implementation if its initial performance is unacceptable.

We expect the study of core concepts to still be relevant even if the main exploitation of parallelism takes place in libraries written by specialists, as some foresee. For the same reason that model curricula currently include linked lists, binary trees, and other elementary data structures, programmers in the future will need some understanding of the implementation of parallel libraries in order to use them effectively. A firm understanding of core concepts also forms the foundation for more performance-focused programmers to develop their expertise. Just as some areas such as HPC and games require greater performance optimization in sequential systems, we expect some programmers to strive for higher levels of parallel performance. Note that we are not hostile to programmers squeezing all possible performance from the silicon; we just see an important distinction between this and concurrent programming as it should be introduced to *all* computer scientists.

The other change we see as necessary to adapt an HPC curriculum into one aimed at a broader group of students is the development of appropriate examples and assignments. HPC is focused primarily on scientific applications, a fact that is strongly reflected in textbook presentations of concurrent programming. Typical applications used as examples are matrix operations (matrix-vector or matrix-matrix multiplication, LU decomposition, etc), numerical integration, finite element computations, and FFTs. Although important, these applications are not intrinsically motivating to students without a background in computational science. This lack is particularly apparent to us in our classrooms at liberal arts colleges, where limits on the number of courses we can require mean that our students often do not take as many science courses as in engineering programs. Even in engineering programs, however, strong prerequisites for concurrent programming would force it to remain a specialized area. To expand the study of concurrent programming out of advanced courses, it must be presented using applications with broad appeal.

Of course, finding applications with broad appeal is easier said than done, but a couple of nice examples are known. Several textbooks talk about game tree evaluation, tapping into the eternal popularity of AI. Another good example in use is generating the Mandelbrot set, a simple computation with a visually-appealing result. (Other fractals would also be appropriate.) Other simple but visually-appealing examples can be taken from the graphical approaches being developed for introductory CS (e.g. media computation [6]). More advanced examples from graphics would be ray tracing or animation rendering. Another area of exciting applications is using tools like MapReduce on large data sets to do collaborative filtering or other socially-based computations. We view expanding this list of appealing examples as a challenging but important task.

With the rise of multicore hardware, it is crucial that our graduates achieve basic proficiency with concurrent programming. To accomplish this, we need to keep in mind the goal, which is to prepare students to write programs that scale reasonably well, rather than focusing entirely on peak performance. Students need a basic understanding of key issues of concurrency, including an intuitive "algorithmic taste" as well as the tools for aggressive optimizations when these are needed. We see the need to develop teaching materials on concurrent programming that are specifically designed to reach a broad audience, using applications that resonate with students. This will necessarily be a group effort and we look forward to being part of the discussion.

REFERENCES

[1] G. Blelloch, "NESL: A parallel programming language," http://www.cs.cmu.edu/ scandal/nesl.html, viewed Aug 2009.

[2] Sun Microsystems Laboratories, "Project Fortress community," http://projectfortress.sun.com/Projects/Community, viewed Aug 2009.

[3] Apache Software Foundation, "Welcome to Hadoop MapReduce!" http://hadoop.apache.org/mapreduce/, viewed Aug 2009.

[4] A. Bhatele, "PPL: Parallel objects," http://charm.cs.uiuc.edu/research/charm/, viewed Aug 2009.

[5] M. Frigo, "The Cilk project," http://supertech.csail.mit.edu/cilk/, viewed Aug 2009.

[6] M. Guzdial and E. Soloway, "Computer science is more important than calculus: The challenge of living up to our potential," *Inroads-The ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 5–8, 2003.

AUTHOR BIOGRAPHIES

David Bunde is an Assistant Professor of Computer Science at Knox College, a liberal arts school with three CS faculty and a major requiring ten CS courses. He is supported in part by Howard Hughes Medical Institute grant 52005130. He has taught small units on GPU programming, Hadoop/MapReduce, and multi-threaded programming in appropriate courses. This Winter, he will teach a concurrent programming course.

Jens Mache is currently an Associate Professor of Computer Science at Lewis & Clark College, a small liberal arts school with three CS faculty. For a CS & math major, students take six CS courses. In 2008, a CS major requiring nine CS courses was added. Jens has twice taught an undergraduate-only course on parallel and cluster computing. In addition, he has spent eleven summers mentoring undergraduates on collaborative research projects that often involve parallelism. He is supported in part by NSF grants 0649068, 0720914 and 0411237.