Concurrency in the Curriculum: Demands and Challenges

Position Paper

Curtis Clifton
Dept. of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana USA
clifton@rose-hulman.edu

ABSTRACT

This position paper describes my background as a practitioner and educator. It outlines some ideas on the general directions for curricular change to address concurrency and parallelism. Finally, the paper identifies a key challenge in making this transition.

1. BACKGROUND

I learned to program in Applesoft BASIC as a primary school student in the early 80s. Undergraduate studies in Electrical Engineering exposed me to FORTRAN and Pascal. As an intern at IBM I picked up APL and C. Then I embarked on a career as a controls engineer and found that ladder logic on programmable logic controllers (PLCs) was the development platform of choice at the companies where I worked.

After 6 years in industry, I decided to pursue a career in teaching and research, so returned to school. In my first year of graduate school I was introduced to Scheme, SmallTalk, Haskell, Java, and C++. Despite exposure to all these languages, plus a few left unlisted, I only did a single academic project that involved concurrency in any real way, a project in an Operating Systems course that used fork and join. Unfortunately, this lack of exposure to practical concurrency and parallelism is not atypical.

I'm now in my fifth year as an assistant professor at Rose-Hulman Institute of Technology teaching software development, programming languages, and compilers to small classes of bright students. I've taught all the courses in our introductory sequence, including object-use-early CS1 (first in Java, then Python), CS2 (Java), and Data Structures (Java). I also developed and teach an upper-division elective course, Programming Language Paradigms, where we explore three different programming languages through a series of programming etudes [2] focusing on particular features of each language.[1] I try to include at least one language focused on concurrency. Thus far I've used Erlang. Students also conduct a team project in a fourth language. I plan to encourage at least one team this fall to explore the Hadoop framework for data-intensive scalable computing.

I come to the workshop as an outsider to the fields of parallel and scientific computing. I recognize the significance of the transition to multicore computing and am seeking out the best ideas for how to prepare our students for this new age. I attended a three day, NSF-sponsored workshop on Data Intensive Scalable Computing at the University of Washington-Seattle in 2007. I also attended the birds of a feather session on Multi-Core Programming in the Curriculum at the SIGCSE Technical Symposium on Computer Science Education in 2009. Last spring I led a discussion with our department's industrial advisory board on what curricular changes they think are needed to address the transition. Given the unsettled nature of the technology, their responses were wide-ranging, from just teaching traditional locks and threads to completely integrating concurrency and parallelism across the curriculum.

Rose-Hulman is a small, teaching-focused school. Our mission is centered on outstanding undergraduate education. Our teaching focus and mission lead to small class sizes (fewer than 25 students) and close interactions between students and faculty. These generally serve our students well. However, in times of transition we must work hard to not be caught flat footed. None of our 12 faculty members is engaged in research on parallel or multicore computing. We don't have experts on the cutting-edge science, but we do have expertise on active and project-based learning. We haven't yet made significant changes to our curriculum to address the growing importance of concurrency and parallelism. I'm working to gather information about best practices and will begin a discussion this fall about how best to adapt.

2. PRESENT AND FUTURE DEMANDS

As an outsider to the current research, I don't bring strong preconceived notions about how best to prepare undergraduate Computer Science students for the age of concurrent and parallel programming. My sense is that we aren't ade-

Submitted to the *First Workshop on Curricula in Concurrency and Parallelism*, OOPSLA 2009

[1]Our students take a required Programming Language Concepts course taught using the *Essentials of Programming Languages* [3] pedagogy—a series of interpreters implemented in Scheme.

quately preparing students for this fundamental transition, in part because we don't yet understand how software will be developed for these systems.

Certainly software is already being developed for such systems, but it doesn't seem like the low-level techniques of message passing libraries and threads will scale. By "scale" here I don't mean scaling to more processors, but scaling to more programmers. Implementing software using these low-level techniques is notoriously hard. I suspect that advances in programming languages, compilers, and operating systems will change the techniques used to develop for massively multicore and distributed systems. Given that, we're faced with two complementary demands:

- we must prepare our students to develop software in the present, using the current tools and techniques; and

- we must prepare our students for the future, to think about issues of currency and parallelism so that they are ready to adopt the new tools and techniques as they appear.

In the workshop call, the organizers are asking the right questions to get at the curricular effects of these two demands.

To be prepared for the future, every Computer Science graduate should understand the fundamental ideas of race conditions, deadlock, and the overhead involved in parallel algorithms. They should understand how our traditional techniques of algorithm analysis are inadequate not only in the presence of memory system effects, but also in the presence of parallelism. To be prepared for the present, they should be able to design and implement scalable software that takes advantage of parallelism using current techniques.

We will have the greatest success teaching concurrency and parallelism by combing the top-down and bottom-up approaches. Different students learn best in different ways. Some need to understand the nitty-gritty details to accept the big picture. Others need the big picture to motivate learning the details. Besides the differences in students, the two demands above also argue for using both teaching approaches. Practically, low-level approaches are in common use and high-level approaches—exemplified by MapReduce and open implementations like Hadoop—are growing in use. In the future we don't know what approaches will dominate.

The questions of whether to teach sequential programming as a special case of concurrent and parallel programming and whether these issues should be addressed in introductory computer science courses are intimately related. As Joe Armstrong says in *Programming Erlang*, "the world is parallel," and "A deep understanding of concurrency is hardwired into our brains."[1] Our students, particularly those who come to us without prior programming experience, already know how to think concurrently. We often struggle to teach them to think sequentially, only to later teach them how to apply more complicated reasoning atop that sequential base to manage concurrency.[2] This leads to a fragile epicycles-on-epicycles mental model. We should teach concurrency early and treat sequential programming as the special case. Like elliptical orbits, this is seemingly more complicated, but the result is a simpler, more robust model of reality.

[2]Our emphasis on imperative languages in the early curriculum also contributes to our challenges teaching concurrency and parallelism.

Our department is taking a mixed approach on the question of concentrating or distributing these topics in the curriculum. We are developing a new systems course for freshmen that takes a breadth-first approach to hardware logic, operating systems, and hardware-software interfaces. This course will introduce some topics in concurrency and parallelism. We also introduce thread-based concurrency (using Java GUI applications) in our introductory Software Development sequence. Our junior-level Operating Systems course delves deeper into race conditions, deadlocks, and other pitfalls. We offer a couple of electives that give students more depth, including one on Parallel Computing and my Programming Language Paradigms course. The Institute is inaugurating an interdisciplinary minor this fall on Computational Science. The leader of this effort is based in the Math department and is focusing on traditional scientific computing techniques. Topics like cloud computing, using GPUs for general purpose processing, massively multicore computing, parallel data structures, and analysis of parallel algorithms don't yet have a home in our curriculum.

3. A KEY CHALLENGE

A key challenge faced by small departments, and by any departments looking to distribute concurrency and parallelism ideas across the curriculum, is the lack of expert faculty. This problem was also faced when introducing object-oriented software development into the curriculum. The community must help provide resources for teaching these topics. These resources might include concise, complete reading material targeted at an undergraduate audience, on-line lectures or tutorials, and well written exercises and project assignments. Training resources for instructors, such as the OOPSLA Educators' Symposium, are also needed.

4. CONCLUSION

The transition to massively multicore and distributed computing is already well underway. I look forward to sharing in the discussion at the workshop, bringing new curricular ideas back to my home campus, and participating in the continuing efforts of the educational community to prepare all our students for this new age.

5. REFERENCES

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Programmers, 2007.

[2] J. Bergin. Variations on a polymorphic theme: An etude for computer programming. In *Ninth Workshop on Pedgogies and Tools for the Teaching and Learning of Object Oriented Concepts*, 2005.

[3] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.