Early and often: Bringing more parallelism into undergraduate Computer Science curricula

Richard Brown
St. Olaf College (USA)
rab@stolaf.edu

Elizabeth Shoop
Macalester College (USA)
shoop@macalester.edu

Joel Adams
Calvin College (USA)
adams@calvin.edu

Curtis Clifton
Rose-Hulman Institute of Technology (USA)
clifton@rose-hulman.edu

Mark Gardner
Virginia Tech University (USA)
mkg@vt.edu

Michael Haupt
Hasso-Plattner-Institut
University of Potsdam (Germany)
michael.haupt@hpi.uni-potsdam.de

1. The time is now

In view of recent industry shifts towards both multi-core processors and applications of distributed computing through techniques such as map-reduce, the question naturally arises: how can Computer Science (CS) undergraduate programs respond with curricular changes to prepare their students for the future of computation, in which parallelism and concurrency will be a necessity, not an option? Up to now, the innovations in hardware parallelism that have driven advances in processor design have been transparent to programmers and programming environments. This is no longer the case. As hardware designers turn to ever increasing numbers of cores, software designers must explicitly turn to parallelism (and concurrency – for convenience, we will use the term *parallelism* to include both) to take advantage of these cores. If not, the steady advances in performance that software designers formerly received for free will cease [9][10][7].

As a community, we have reinvented the CS curricula in the past when we saw the need. As the renaming of this conference indicates, Object-Oriented Programming has now become a mainstream component of undergraduate programming. This change evolved slowly: more than 20 annual meetings of OOPSLA took place before that systemic change could fully appear. In contrast, we need increased parallelism in our courses immediately: quad-core chips now appear in commodity computers, and 8-core chips are already in mass production.

We were members of an international ITiCSE 2010 Working Group who considered strategies for changing undergraduate CS curricula at all types of institutions to effectively and expediently respond to this challenge. The main product of our Working Group was a report analyzing the needed changes and suggesting how to achieve them. In this position paper, we summarize our proposed approach to curricular change and indicate some examples.

2. What CS educators can do

Our report [1] suggests a framework for a body of knowledge of parallelism, illustrated in Table 1, and describes the knowledge areas identified in that framework. We have developed a set of essential learning objectives for each knowledge area, which should serve as a guide when incorporating parallelism topics into courses.

For each knowledge area, we describe a set of central ideas that CS graduates should understand. For example, in the area of conceptual issues and theoretical foundations, we identify scalability, speedup, and efficiency as essential concepts, among others. For software design, we describe known patterns of parallel programs students should consider when developing parallel programs. In data structures and algorithms, we provide ideas for studying shared access to data structures, with or without the use of locks, and identify parallel algorithms that might be introduced. In the area of software environments, the central ideas include models of parallel computation (such as the shared memory model and the actor model) and we provide information on the numerous languages and software libraries that support such models. In the hardware area, we argue that exposing students to a variety of hardware topics (e.g., MIMD, SPMD, SIMD, shared vs. distributed memory) helps them to develop adequate conceptual models of hardware, informing choices they make in the other parallel knowledge areas.

Table 1. Organizing the body of knowledge in parallelism.

<table>
<tr><td rowspan="4">Motivating Problems and Applications</td><td>Software Design</td><td rowspan="4">Conceptual Issues and Theoretical Foundations</td></tr>
<tr><td>Data Structures and Algorithms</td></tr>
<tr><td>Software Environments</td></tr>
<tr><td>Hardware</td></tr>
</table>

Our report continues to propose teaching and learning strategies for introducing more parallelism in CS undergraduate programs. Of course, discussions of parallelism and concurrency have been staples at most institutions, in courses such as Operating Systems, Computer Architecture, and perhaps a dedicated parallel computing course. Since all CS students must now become more acquainted with more parallelism, we argue for expanding the range of courses that include parallelism. We assert that a *spiral and experiential ("hands-on")* approach to learning the principles and concepts of parallelism will instill in our students the abilities that they will need throughout their careers: to "think in parallel" and adapt to new CPU features and developments. Here, we refer to the Spiral Principle in the European discipline of Didactics of

Informatics, in which a student revisits notions periodically at increasing depth and complexity [3][4].

For example, as CS educators, we no longer serve students well if we portray programming as purely sequential in our courses. Sequential programming skills remain crucial, since they are applied in most parallel programming strategies. But sequential programs will not scale up when they are ported from single core machines to multi-core machines. To achieve faster performance on new computers, such sequential computations must be replaced by parallel computations, which means students must be trained to design programs with parallelism in mind. Frequently presenting and returning to parallelism throughout a student's curriculum will provide an invaluable sense of context, as well as useful skills and knowledge of the concepts and principles of parallel computation. In short, we advocate teaching parallelism "early and often" at all levels of an undergraduate CS curriculum.

We identify examples in our report of strategies for inserting notions of parallelism with hands-on exercises that require little or even no extra time in a course's syllabus. While avoiding assumptions about a particular institution's curricular design, we suggest strategies for incrementally introducing parallelism in introductory courses and courses treating various intermediate and advanced subjects, such as data structures, software design, algorithm analysis, and programming language concepts. For example, we cite examples that we and others have used in practice at the introductory level for getting students thinking about data and task parallelism program designs and implementing their first parallel programs [2][5][6][8]. We also provide examples and references for adding parallel algorithms, parallel access to data structures, and patterns of parallel program design to intermediate courses in any curriculum.

In our report, we also observe that certain languages used for teaching programming in a CS curriculum also offer opportunities for treating parallelism. We found many rich opportunities for students to explore parallelism in the context of programming languages throughout our working group process, largely during the months of electronic meetings that preceded the drafting of our report. We provide examples of languages that have been designed with parallel programming in mind, including the introductory systems Alice and Scratch, functional languages such as Haskell and Erlang, and multi-paradigm languages such as Scala and Fortress. We also suggest ways that additional libraries for traditional languages could also be used to enable students to practice thinking in parallel to solve problems.

3. Examples of change

The "early and often" approach we advocate and/or other elements of our report appear in several existing initiatives to teach more parallelism in undergraduate CS. One publicly accessible example is a joint effort at Macalester and St. Olaf Colleges, in which modular teaching materials each capable of being inserted in a variety of course settings are being developed and tested in courses at both institutions at all curricular levels. The effort includes supplementary software and documentation for a variety of parallel computation platforms to support "hands-on" exercises [5].

Our report has already formed the basis for curricular reform efforts at Rose-Hulman Institute of Technology. Department faculty there read a draft of the report and agreed to use the continual improvement process for their ABET accreditation as the lever to effect change. The faculty agreed to revise the program outcomes for both Computer Science and Software Engineering to explicitly mention scalability. The department focused on scalability because that notion seemed concisely to capture the essence of what is needed to take advantage of multi-core computing. The current draft program outcome states: "By the time students graduate with a computer science degree from Rose-Hulman, they will be able to identify scalable solutions to new problems and analyze the scalability of existing solutions." The faculty then reviewed the full set of courses offered by the department in light of this change. As a result, nearly half of the department's courses will incorporate scalability and parallelism as this process moves forward. The individual course outcomes will touch on the various knowledge areas identified in the report, filling in details around the broad strokes of the Institute's new program outcomes.

We offer our comments as a starting point for discussions of the urgent challenge of injecting parallelism into CS curricula, and we seek feedback from the SPLASH community on these ideas.

4. References

[1] Brown, R., Shoop, E. et al. 2010. Strategies for Preparing Computer Science Students for the Multicore Worl. *ACM SIGCSE Bulletin*. under review, (2010).

[2] Bruce, K.B., Danyluk, A. et al. 2010. Introducing concurrency in CS 1. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 224-228.

[3] Bruner, J. 1974. *Toward a Theory of Instruction*. Belknap Press of Harvard University Press.

[4] Bruner, J. 1977. *The Process of Education*. Harvard University Press.

[5] Ernst, D.J. and Stevenson, D.E. 2008. Concurrent CS: preparing students for a multicore world. *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (Madrid, Spain, 2008), 230-234.

[6] Garrity, P. and Yates, T. *WebMapReduce*.

[7] Larus, J. 2009. Spending Moore's dividend. *Commun. ACM*. 52, 5 (2009), 62-69.

[8] Parallel Computing in the Computer Science Curriculum. *http://csinparallel.org*. Accessed: 08-03-2010.

[9] Sutter, H. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*. 30, 3 (2005), 202-210.

[10] Sutter, H. and Larus, J. 2005. Software and the Concurrency Revolution. *Queue*. 3, 7 (2005), 54-62.