A Modular Approach to Teaching Parallelism at All Levels of CS Curricula

Richard Brown
Mathematics, Statistics, and Computer Science
St. Olaf College
Northfield, MN
rab@stolaf.edu

Elizabeth Shoop
Mathematics, Statistics, and Computer Science
Macalester College
Saint Paul, MN
shoop@macalester.edu

Abstract

For the foreseeable future reliable hardware speedup will only be delivered by increasing the number of *cores* (circuits for carrying out instructions) per computer. This means that our CS undergraduate students need to learn more parallelism now, in order to be prepared for careers that will demand increasing knowledge and practice of parallel computing. We can no longer delegate study of parallelism and concurrency to hardware architecture and operating systems courses, because now knowledge of parallel algorithms and thread-safe data structures will be factors in ordinary high-level software. There is an additional demand for services that require data-intensive scalable computing, which often require distributed-computing parallelism. In order to create a necessary transition towards teaching more parallelism in undergraduate CS curricula, we propose an incremental approach to adding principles and practices of parallelism to existing CS courses. This calls for modular teaching materials requiring perhaps a couple of class days to present, designed for flexible application to a variety of courses that may appear in many institutional and curricular contexts. To date, we have produced several flexible teaching modules (available at `csinparallel.org`) for courses ranging from CS1 to upper-level courses in algorithms, programming languages, etc., together with supporting materials and hands-on exercises involving appropriate software and hardware tools. The applied exercise assignments can be deployed on local standard systems (e.g., Java threads), or on established and reliable external systems (e.g., cloud computing services or Intel's Manycore Testing Lab). We are currently in the process of class-testing and formally assessing these materials, and have plans to create several more modules.

1 Introduction and Motivation

In 1965, Intel co-founder Gordon Moore observed that the number of components in computer circuits had doubled each year since 1958, and famously predicted that this doubling trend would continue for another decade. Incredibly, Moore's "Law" has now held for over five decades. Until about 2005, this doubling could be applied to improvement strategies in computer architecture that directly increased the performance of sequential (non-parallel) programs, such as higher clock speeds and deeper micro-architecture pipelines for carrying out machine-level instructions. But as Herb Sutter of Microsoft observed that year [9], those strategies can no longer be sustained. Consequently, the performance of sequential programs will remain static over time, and the ongoing exponential improvements in hardware performance will only make a difference for programs that take advantage of parallel computing.

The imperative of parallelism arises from engineering constraints in computer design, not from corporate strategy. A 2006 white paper released by a study group at the University of California at Berkeley identified three "walls" in the design of computers (involving power management, memory access speed, and instruction-level parallelism) that together constitute an insurmountable barrier to continued performance improvement of individual computing *cores*, i.e., circuits for carrying out instructions [2]. As a consequence, the only way to maintain the exponential pattern of hardware performance improvement is to increase the number of cores per CPU. As Michael Wrinn of Intel says, "We don't make sequential computers anymore" [10]. The Berkeley team predicted that the number of cores per processor will grow to dozens, hundreds, even thousands. Indeed, eight-core CPUs have been available for purchase since last year, and an experimental chip with 48 Pentium cores was announced 18 months ago [8]. As the number of cores per CPU continues to increase, the performance of sequential programs will soon pale in comparison with programs that employ parallel computation.

Multi-core computer design mandates an inevitable shift to parallel computing, but other developments in the computing industry offer powerful optional opportunities through parallelism. In particular, systems such as Google's proprietary MapReduce system [7] and Apache's Hadoop [1] for *data-intensive scalable computing (DISC)* make it feasible to compute effectively with terabytes and even petabytes, using distributed computing on affordable clusters. (Note: As a verbal convenience, we will treat *parallelism* and *parallel computing* as generic terms, encompassing hardware parallelism within a single computer, distributed computing involving multiple computers, and software concurrency.) With the clusters typically implemented on cloud computing resources, DISC has launched a new era of web services, and holds great potential for computations on large data that may have nothing to do with web services.

Given the "carrot" of new opportunities presented by today's powerful parallel computing strategies, along with the "stick" of the long-term reality of more and more cores per computer, our undergraduates in CS must learn more principles and practices of

parallelism or risk graduating and entering the work force without the foundation in parallel computing they will need during their careers.

2 A Parallelism Body of Knowledge

If our students need to learn more parallelism, what shall we teach them? We are fortunate that the body of knowledge of parallelism is well understood, based largely on fundamental research carried out in prior decades, and further developed through the long application experience of the scientific computing community and others.

We recently participated in an international study [4] that suggested a framework for this body of knowledge of parallelism, illustrated in Table 1, and described the knowledge areas identified in that framework. Our working group developed a set of essential learning objectives for each knowledge area, to serve as a guide when incorporating parallelism topics into courses. We also described a set of central ideas that CS graduates should understand for each knowledge area. Examples in each knowledge area indicate this approach: the central ideas in the area of *conceptual issues and theoretical foundations* include scalability, speedup, and for students to consider when developing parallel programs; in *data structures and algorithms*, students benefit from studying shared access to data structures, with or without the use of locks, and from an introduction to parallel algorithms; 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 information on the numerous languages and software libraries that support such models is provided; and in the *hardware* area, 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>
<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>

Table 1. Organizing the body of knowledge in parallelism.

3 A Strategic Plan For Teaching More Parallelism

Our working group [4] also addressed the strategic question of how to teach more of the body of knowledge of parallelism in existing CS curricula. While adding courses in parallelism is certainly useful and may be the easiest approach in some local circumstances, 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, our group referred to the *Spiral Principle* in the European discipline of Didactics of Informatics (comparable to Computer Science Education in the U.S.), in which a student revisits notions periodically at increasing depth and complexity [5][6].

For example, our group held that CS educators no longer serve students well if they 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, our working group advocated teaching parallelism "early and often" at all levels of an undergraduate CS curriculum.

In order to implement the goals of presenting elements of the parallelism body of knowledge in a spiral, hands-on approach, *we propose here an incremental strategy for inserting parallel computing notions, through short, flexible, well-supported teaching modules at all undergraduate levels that can be used in a wide variety of courses and curricula.* Our project promotes this strategy by producing examples of teaching modules with those qualities, providing platform-specific support for the computational resources needed for hands-on computational exercises for those modules (*Parallel Platform Packages*), and encouraging the community of CS educators to teach more parallelism through web resources such as csinparallel.org, which provides our modules and platform supports and enables others to contribute commentary and/or further modules.

We design our modules with the following goals in mind:

- Compatibility with many institutional approaches to teaching CS;
- Flexible for use in many potential courses (e.g., modules with support for multiple languages or parallelism libraries);
- Requiring only incremental change in a course's syllabus;
- As simple and quick as possible to deploy in a course; and
- Minimal time investment required of instructors who seek to deploy those modular materials, including instructors who are *not* specialists in parallel computing.

We choose this incremental, modular approach to adding more parallelism to CS curricula for a number of reasons. For many professors and institutions, adding a new course is difficult, and replacing an entire course or a large segment of course material is very difficult or prohibitive, whereas a one- or two-day module represents an affordable change for those with a commitment to inserting parallelism into undergraduate CS education. Of course, more of the parallelism body of knowledge will surely appear in

textbooks soon. However, the publication cycle for textbooks is lengthy, and a modular approach can be delivered to professors much more quickly. Since our modules seek independence of particular texts, syllabi, approaches, and curricula, professors can apply them in a broad array of contexts, including educational scenarios beyond those originally envisioned for those modules. Perhaps the most compelling argument for our modular approach is that the techniques of parallel computing will develop rapidly in the coming years, as new programming models and software tools emerge in response to the need and opportunity of parallelism [3]. We promote teaching CS undergraduates the principles of parallel programming reinforced by hands-on experience that is informed by current practice; yet current practice is a moving target, and a matter of current research. The online distribution of our modules supports quick deployment and instant update, making the modular approach flexible and responsive to new developments and interests.

4 Modules for teaching parallelism

The csinparallel.org modules offer a number of features that support their proper use and adoption, typically including:

- Explicit learning objectives for each module;
- Suggested contexts for use of each module;
- Readings for students to prepare for class presentations and/or exercises;
- In-class or laboratory activities;
- Follow-up homework exercises;
- Instructor support materials;
- Assessment tools; and
- Opportunities for discussion, feedback, and support of those modules.

In addition, the Parallel Platform Packages provide options for experiential learning activities and exercises in various modules.

The modules fall into two categories. *Application/activity* modules explore parallel computing principles and practices in a specific direction, and include computational exercises using parallelism. *Concept modules* present fundamental or prerequisite notions needed by multiple other modules, or key topics in parallelism that may not (yet) naturally fit in other modules. Most modules fit in the application/activity category; only one concept module has been produced to date.

About halfway through our two-year project, we have produced versions of nearly a dozen modules in various states of completion, including the following:

- Map-Reduce Computing using WebMapReduce (intended for CS1)
- Concurrent Access to Data Structures in Java
- Multicore Programming with Intel's Manycore Testing Lab
- Introduction to Parallel Computing Concepts (concept module)
- Parallel Sorting Algorithms

- Intermediate Map-Reduce Computing
- Concurrency and Map-Reduce Strategies in Programming Languages

We will elaborate on some of these modules in order to describe our approach more fully. Additional information and the modules themselves may be found at csinparallel.org.

4.1 Map-reduce computing using WebMapReduce

Learning goals:

- Students should be able to identify basic forms of data parallelism in computational problems.
- Students should be able to distinguish between sequential and parallel computation, and identify the practical significance of each.

Brief description: Intended for introductory CS courses that involve programming, with support for several common introductory languages (Java, Python, C++, Scheme).

Discussion: This module introduces data-parallel computation (single program applied in parallel to multiple data sets) and related concepts, with programming exercises to carry out map- reduce computations using the WebMapReduce (WMR) platform package (see section 2.3). Expository teaching materials for students motivate map-reduce computing, citing both prominent applications in industry and potential for use in interdisciplinary student project that may involve large data, then present map- reduce computing with examples in a particular introductory language. Programming exercises reinforce the concepts presented in the reading, then proceed to explore practical features of WMR that could be useful for student projects. Non- programming exercises explore additional concepts related to parallelism, such as multi-tasking.

This is the first module we developed. We have taught prototype versions of this module in several versions of our introductory courses at both of our institutions, using Python and Scheme languages, typically as a two-day unit. Note that WMR is extensible to other choices of programming language, and the language interface can be tailored to local conventions and practices.

4.2 Intermediate Introduction to Parallel Computing

Learning goals:

- Given a description of a problem to solve, students should be able to discriminate whether it requires a data parallel or task-parallel solution

- Given a description of a parallelizable algorithm, students should be able to apply the principle of Amdahl's law to determine the maximum speedup of that algorithm using N processors.

- Given a language-specific library and an associated back-end platform, students will be able to implement at least one simple algorithm with data parallelism and at least one simple algorithm using task parallelism

- Students will be able to discriminate DISC from other forms of parallel computation, describe the operation of the map-reduce strategy for DISC in general terms, and list significant applications of DISC in industry.

- Given an example of parallel code, students will be able to identify race conditions and describe how to remove those race conditions using a synchronization primitive

Brief description: Provides general background in parallel computation that is common to one or more subsequent modules. Programming language-independent; intended for students in a second or later CS course.

Discussion: We consider this to be a *concept module*, whereas most of the other modules listed are *application modules*. This Intermediate Introduction module gathers concepts of parallel computing that other modules require, including terminology, a discussion of speedup (including Amdahl's Law), communication options, and conceptual issues such as fault tolerance and mutual exclusion. At this writing, exercises are under development.

4.3 Parallel Sorting Algorithms

Learning Goal:

- Given a theoretical Parallel Random Access shared memory model (PRAM), a particular memory access model, and an interconnection network, students should be able to develop an algorithm for sorting a large collection of N data items.

Brief description: Suitable for use in an algorithms course; uses background material from the Intermediate Introduction (section 4.2).

Discussion: This is in part a concept module that describes how to reason about and design various parallel versions of merge sort. It begins with an introduction to the theoretical PRAM machine as the basis for formulating parallel algorithms (a technique that haslong been used in the study of parallel algorithms). It is designed to be expanded to include other sorting techniques, such as quicksort, and includes exercises.

5 Conclusion and Future Work

Recent developments, including the advent of multi-core computers and the available potential of data-intensive scalable computing on distributed clusters, indicate that CS

students will in inevitably encounter parallelism in their careers as computing professionals. We propose an incremental strategy to adding more principles and practices of parallel computing at all levels of CS curricula, in support of a spiral, hands-on approach to teaching parallelism. Our approach calls for modular teaching materials that professors and CS programs can use with a variety of courses and local curricular choices, supported by learning objectives and other educational context, with module exercises facilitated by Parallel Platform Packages to assist with deployment on particular computing platforms and parallelism libraries. We present features for such modules, and discuss examples among the modules we have produced to date.

Acknowledgment

This work was supported by a joint NSF CCLI grant DUE- 0942190/0941962. We would like to thank our undergraduate research collaborators, including Patrick Garrity '12, Maura Warner '12, Mitchell Wade '11, and Timothy Yates '12, as well as our colleagues at both St. Olaf and Macalester for generously making it possible for us to test our materials in their courses.

References

[1] Apache Hadoop Project. *http://hadoop.apache.org/*. Accessed: 06-28-2010.

[2] Asanovic, K., Bodik, B. et al. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report #UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

[3] Asanovic, K., Bodik, R. et al. 2009. A view of the parallel computing landscape. *Commun. ACM*. 52, 10 (2009), 56-67.

[4] Brown, R., Shoop, E. et al. 2010. Strategies for Preparing Computer Science Students for the Multicore Worl. *Proceedings of 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)* (2010), to appear.

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

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

[7] Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. *OSDI* (2004), 137-150.

[8] HPCwire: Intel Unveils 48-Core Research Chip. *http://www.hpcwire.com/features/Intel-Unveils-48-Core-Research-Chip-78378487.html*. Accessed: 06-27-2010.

[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] Wrinn, M. 2010. Suddenly, all computing is parallel: seizing opportunity amid the

clamor. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 560-560.