The Role of Programming Languages in Teaching Concurrency

Kim B. Bruce
Pomona College

Stephen N. Freund
Williams College

Doug Lea
SUNY Oswego

September 4, 2009

1 Introduction

By common conventions, *parallel* programming focuses on the exploitation of multiple execution agents to improve performance, and *concurrent* programming focuses on the coordination of multiple independent activities as they arise for any reason. Multicore, multiprocessor, and/or clustered platforms are becoming the *only* platforms available for executing programs. Thus, Computer Science education increasingly entails education in parallel and concurrent programming. We expect that over time, parallelism and concurrency will be infused in the curriculum. For example, useful parallel algorithms and concurrent data structures will be taught alongside classic sequential ones. Similarly, development of correct, reliable, and efficient concurrent and parallel programs will become commonplace topics in systems and engineering courses.

In this position paper we restrict attention to a basic prerequisite to effective integration. The study of programming languages provides a basis for teaching students how to express concurrent and parallel constructions, how to understand their semantics, and how to analyze properties of their programs. More concretely, we present recommendations stemming from recent SIGPLAN-sponsored activities regarding programming language curriculum development. As a first, but important step, toward best preparing students for the increasingly-concurrent landscape, we advocate their recommendation to include a common core of language and implementation concepts in all Computer Science and Software Engineering programs.

The ACM/IEEE CS Computing Curricula 2001 report [CS01] contains scant mention of concurrency and parallelism. There is a 6 hour knowledge unit OS/Concurrency, but that unit is more concerned with the support of concurrency in operating systems than with the construction of concurrent programs. The Curriculum 2008 interim report [Int08] expresses significant concern about how to address the growing importance of concurrency but makes no changes in the core curriculum to reflect this trend.

An increasing number of computer science and computer engineering programs provide advanced electives in parallel programming and concurrency. However, it is our opinion that *all* undergraduate students need a substantial introduction to concurrent programming as it is used in high-level languages. As a simple example, event-driven programs in which users may trigger long-running actions, such as running an animation, require separate threads in order to maintain responsiveness to later user actions. More importantly, as we move to many-core computers, programmers will need to understand how to write programs that can take advantage of the multiple processing units available on virtually all new computers.

In this position piece we discuss how programming languages related content can play an important role in teaching undergraduates about concurrency.

2 Programming Languages & Concurrency

The First SIGPLAN Workshop on Undergraduate Programming Language Curricula was convened in May of 2008 by SIGPLAN officers Kathleen Fisher and Chandra Krintz to discuss the role of programming language design, implementation, and application in modern undergraduate computer science education. At the workshop and in further e-mail communications, the group formulated recommendations [ABB+08] on

what programming languages topics and experiences every computer science undergraduate should know or have before graduation.[1]

The report presents three overreaching objectives for the study of programming languages in the undergraduate curriculum.

1. Quickly learn programming languages, and how to apply them to effectively solve programming problems.
2. Rigorously specify, analyze, and reason about the behavior of a software system using a formally defined model of the system's behavior.
3. Realize a precisely specified model by correctly implementing it as a program, set of program components, or a programming language.

In the rest of this position piece, we summarize those recommendations as they relate to concurrency and present a few guidelines for how they may be realized in courses. Of the forty hours of "core" programming languages material in the recommendations, five hours are devoted to concurrency and parallelism, with another two hours recommended if time is available. While the report organizes the recommendations by which of the objectives they satisfy, we choose a somewhat different organization here to make the content and motivation more clear.

2.1 Parallelism

The report recommends 2.5 to 3.5 hours of lecture on "deterministic" parallelism, in which multiple, but non-interacting, computations may execute at the same time. Typically, the best vehicle for this coverage is presentation of parallel map, reduce, and apply-to-all constructs that extend those presented in the context of functional programming. (If time is available, the report recommends also covering speculative techniques such as futures.) By understanding the use of constructs statically precluding data races and atomicity violations, students also come to learn the contexts in which they apply (including stream-based and lazy functional programming). They also learn how to use language features to organize computations around multiple simultaneously executing subcomputations to improve throughput.

2.2 Concurrency

The report recommends another 2.5 to 3.5 hours on computational models that permit multiple concurrent activities within a program to communicate nondeterministically. Typically, the best vehicle for this coverage is presentation of shared-memory concurrency constructs, including language-based or library-based thread construction and synchronization mechanisms such as locking that manage safety, liveness, and their associated correctness issues. Additionally, as time permits, presentation of transactional, event-based, actor-based, and/or asynchronous message-passing constructs demonstrates the advantages and disadvantages of alternative approaches to coordination. Because such language features are still a subject of active research, the focus is on the underlying models, rather than on specific languages or features.

3 Conclusion

The parallelism and concurrency topics described in the recommendations of the SIGPLAN Workshop on Undergraduate Programming Language Curricula are essential for undergraduate CS majors. These concepts will play a central role in software development within virtually every area of computing.

Before Curriculum 2001, programming languages were an integral pat of undergraduate curricular recommendations. For example, ACM/IEEE CS Curricula '91 [For91] included 46 hours of core material in

[1]The report assumes that all students gain facility in programming in some language during their first year courses, but intentionally did not address those issues in the report.

programming languages, including 3 hours on distributed and parallel programming constructs. Nearly all of this material, including the material on distributed and parallel computation, disappeared in Curriculum 2001. We believe that this material must be returned to the core, particularly the material sketched above on deterministic parallelism and concurrency.

References

[ABB+08] Eric Allen, Ras Bodik, Kim Bruce, Kathleen Fisher, Stephen Freund, Robert Harper, Chandra Krintz, Shriram Krishnamurthi, Jim Larus, Doug Lea, Gary Leavens, Lori Pollock, Stuart Reges, Martin Rinard, Mark Sheldon, Franklyn Turbak, and Mitchell Wand. Sigplan programming language curriculum workshop: Discussion summaries and recommendations. *SIGPLAN Not.*, 43(11):6–29, 2008.

[CS01] ACM/IEEE CS. Computing curricula 2001. *J. Educ. Resour. Comput.*, 1(3), 2001.

[For91] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991.* ACM Press, 1991.

[Int08] Interim Review Task Force. Computer science curriculum 2008: An interim revision of CS 2001. Technical report, ACM / IEEE CS, 2008.