A Tool-based Approach to Teaching Parallel and Concurrent Programming

Caitlin Sadowski
University of California at Santa Cruz
supertri@cs.ucsc.edu

Thomas Ball
Microsoft Research
tball@microsoft.com

Judith Bishop
Microsoft Research
jbishop@microsoft.com

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Ganesh Gopalakrishnan
University of Utah
ganesh@cs.utah.edu

Joeseph Mayo
University of Utah
u0565813@utah.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

Stephen Toub
Microsoft
stoub@microsoft.com

Abstract

Today, multicore computers are commonplace and university curricula are lagging behind. We need to work concurrency and parallelism into introductory courses, while also maintaining upper-level specialized courses on the topic. Since teachers may themselves require education on the topic, we feel that it is important to make course materials freely available. This position paper outlines some key components we feel concurrency curricula should have, and then discusses how the course materials we are developing satisfies those components.

1. Introduction

Programming for multicore and distributed systems has moved from a fringe activity to a standard practice. These systems have an increasing share in the space of computer hardware, a share which is only likely to increase [2]. However, this shift in programming practice is still impacting university curricula.

Concurrency is difficult to reason about, both in terms of correctness and performance. When writing parallel or concurrent programs, students are faced with an entire new class of potential code bugs. Concurrency bugs are both insidious and prevalent [5, 12].

We believe that:

- High-level abstractions can create a unifying framework for viewing performance with correctness, and can lead to parallel speedups with relatively little effort
- Tools support can make programming faster, less frustrating, and can make programs more correct.

Starting with these tenets, we feel the following components are integral parts to a parallel and concurrent programming curricula.

2. Key Components

Start with abstractions. We agree with several other researchers [7, 8, 16] in advocating starting at a high level of abstraction. In the beginning, parallel and concurrent programming should be taught with a breadth-first, productivity emphasis. Curricula should start by highlighting patterns, not primitives. We should make it easy for students to do high-level parallelism (for example, parallelizing independent loops). It is also very important to initially motivate students with simple examples they can try out that do result in parallel speedups [11].

Later, teach how to navigate abstractions. Unfortunately, performance often depends on factors throughout the entire sequence of abstraction layers. For example, false sharing can lead to unnecessary contention. However, identifying false sharing requires understanding caching behaviours.

Although we feel it is important to start the curriculum at a high level of abstraction, we need to show students how to break those abstractions when they have to. Students need to learn the building blocks of concurrent programming (e.g. threads and volatile variables), and understand what goes on “behind the scenes.”

Emphasize correctness. We believe that university curricula for concurrency should emphasize correctness, although performance issues should not be far behind. Concurrency bugs are notoriously problematic. Other researchers have highlighted the need for students to understand concurrency-specific correctness issues [6, 7, 10, 16]. Beginning students need a solid grasp of how to write correct programs, and what correctness issues are unique to concurrent programs, before performance-tuning their programs.

No more matrix multiply. A focus on appealing examples [7] is important for engaging students with the curriculum.

Don’t discount the most popular model. Threads and shared memory are the dominant paradigm for writing concurrent programs. Students need to be prepared to write programs using this paradigm, and to troubleshoot common types of shared memory bugs. As with other researchers [9], we want students to start leveraging parallelism in programs they write. Realistically, many students will be writing programs in mainstream languages under this dominant paradigm.

Tool support is important. For multithreaded programs, normal unit tests are inadequate for discovering concurrency-specific bugs.

We believe that analysis tools should be used in a classroom setting to identify these bugs [3, 4, 14]. With appropriate tool support, students can build a deeper understanding of correctness issues through experimentation. We believe that this tool-based approach will improve the learning experience by enabling quick and simple experimentation and making debugging less frustrating.

Expose students to new research. There is not a clear consensus on the most effective way to program for multicore systems. Students must be prepared to adapt to changing paradigms [15]. Additionally, students may be engaged by exposure to new research [15].

3. Course

We have been developing a course curricula based on the above convictions, supported by Microsoft technologies [1]. Practical Parallel and Concurrent Programming (PPCP) is a semester-long course that will teach students how to program parallel/concurrent applications, using the C# and F# languages with other .NET libraries. This 16 week (8 unit) course is aimed at beginning graduate or senior undergraduate students. Individual units can also be taught à la carte, and may be sprinkled throughout a computer science curriculum. This course is tilted towards correctness issues, concurrent programming, and shared memory systems. However, a wide breadth of material is covered, including performance pitfalls, message passing, and data parallelism.

PPCP gives students a vocabulary for reasoning about both parallelism and correctness. For example, the DAG model of parallelism presented in Unit 1 provides a general abstraction which can be used to identify both performance bottlenecks and concurrency bugs in applications. Most importantly, the correctness concepts from the course are supported by tools. We have developed an attribute-based concurrency unit testing framework called Alpaca (A lovely parallelism and concurrency analyzer). Students can build understanding of correctness conditions and performance problems through experimentation.

Start with abstractions. The course material starts at a high level of abstraction by introducing patterns (Units 1-4). These patterns take advantage of new .NET 4.0 parallel extensions, and include simple Parallel.For loops, design patterns which avoid data races (e.g. immutability and pipelines), and high-level map and reduce primitives in PLINQ and F#. We emphasize straightforward parallelization of independent computations, plus data partitioning strategies that may lead to independent computations.

Later, teach how to navigate abstractions. In later units (Unit 5 & 8), these abstractions are peeled back to expose the underlying primitives (e.g. threads). Students learn how to build their own thread pool and use low-level synchronization primitives *after* parallelizing independent tasks.

Emphasize correctness. We emphasize correctness throughout the course, along with performance. In each unit, a selection of correctness concepts are highlighted. Furthermore, correctness issues have tool support within Alpaca.

No more matrix multiply. We have included a variety of example applications with the course materials. Many of these have a visual component (e.g. small games).

Tool support is important. In addition to supporting standard unit tests and special performance tests, Alpaca leverages the CHESS stateless model checking framework [13] to run unit tests under different schedules and help mitigate scheduler-dependent nondeterminism in testing. Additionally, dynamic analyses for a variety of concurrency bugs are built-in. In this course, students will regularly analyze their code, including tests of data race detection, deadlock detection, stateless model checking, and linearizability checking. The presence of tool support for concurrency bug detection allows predictable development and testing; for the first time in an academic setting, students have ways to control concurrent unit tests.

Expose students to new research. Throughout the course, we make relatively new research directly available to students. For example, the Alpaca framework builds off of recent advances in stateless model checking and concurrency bug analysis. We use new (.NET 4.0) language features as an integral component of the course. We also have a course module (Unit 8) targeted towards more advanced topics and new models for parallel and concurrent programming.

Acknowledgments

Special thanks to Sherif Mahmoud and Chris Dern for their support.

References

[1] Practical parallel and concurrent programming course materials. `http://ppcp.codeplex.com/`.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[3] M. Ben-Ari. A suite of tools for teaching concurrency. *ACM SIGCSE Bulletin*, 36(3):251–251, 2004.

[4] S. Carr, J. Mayo, and C. Shene. ThreadMentor: a pedagogical tool for multithreaded programming. *Journal on Educational Resources in Computing (JERIC)*, 3(1):1, 2003.

[5] S. Choi and E. Lewis. A study of common pitfalls in simple multi-threaded programs. *ACM SIGCSE Bulletin*, 32(1):329, 2000.

[6] C. Clifton. Concurrency in the curriculum: demands and challenges. In *Workshop on Curricula for Concurrency*, 2009.

[7] J. M. David P. Bunde. Teaching concurrency beyond HPC. In *Workshop on Curricula for Concurrency*, 2009.

[8] D. Ernst. Parallelism is everywhere - so how do we make it accessible? In *Workshop on Curricula for Concurrency*, 2009.

[9] D. Ernst and D. Stevenson. Concurrent CS: preparing students for a multicore world. In *ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2008.

[10] A. Fekete. Teaching students to develop thread-safe java classes. In *ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2008.

[11] D. Joiner, P. Gray, T. Murphy, and C. Peck. Teaching parallel computing to science faculty: best practices and common pitfalls. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 43(3):329–339, 2008.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[14] M. Ricken and R. Cartwright. Test-first Java concurrency for the classroom. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 219–223, 2010.

[15] S. Rivoire. A breadth-first course in multicore and manycore programming. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2010.

[16] M. L. Scott. Making the simple case simple. In *Workshop on Curricula for Concurrency*, 2009.