Ready-For-Use: 3 Weeks of Parallelism and Concurrency in a Required Second-Year Data-Structures Course *

Dan Grossman
University of Washington
djg@cs.washington.edu

1. Introduction and Overview

There is a big difference between believing an undergraduate curriculum ought to expose students to concurrency early and actually doing it — and doing it in a way that non-experts can teach and that fits within other curricular constraints. I argue that a core data-structures course for second-year students (after introductory programming but before more specialized senior-level courses) is an ideal place for a three-week introduction to basic concepts in shared-memory parallelism and concurrency. This basic introduction need not, cannot, and should not produce sophisticated multicore programmers; the purpose is to introduce enough key concepts on which later courses can rely.

More important than this philosophical position is that I have done it: Parallelism and concurrency are now major topics in my department's required data-structures course. Because I am unaware of other curricular resources covering the material in this way — for second-year students in a data structures course using Java — I developed my own. I have packaged these materials, including course notes for students to read, lecture slides, homeworks, a programming project, and sample exam problems, such that they should be usable by others.

This position paper briefly describes the principles, content, and advantages of this approach.

2. Context

With the ubiquity of multicore architectures, many people rightfully want to revise the undergraduate curriculum to introduce concurrency concepts earlier and more often. At the University of Washington, our first systematic major revision of required second-year courses in decades provided a natural opportunity. We modernized our topics while *reducing* the number of required courses so that students can choose paths earlier in the ever-growing field of computing. So a full required ten-week course (we have a quarter system) on concurrency was out of the question given all the other fundamental topics. Three weeks seemed "about right" but that requires integrating the material into a course covering other topics. My position is that our choice of the data-structures course was optimal.

2.1 Where This Material Did Not Fit

Our two-quarter *introductory programming* sequence is large, strong, and stable. Change is difficult, with a large course staff, thousands of students a year, and coordination with community colleges who teach the same course. There is no room for concurrency among basic topics like conditionals, arrays, linked lists, and recursion. It is also unclear these students need to see concurrency: 75% of students in our second course and 88% of students in our first course will not become computer-science majors.

A second-year *software-design course* certainly could deal with concurrency, such as callbacks for GUIs. But relegating the material there gives the impression that concurrency is only a complication rather than a fundamental issue in computing. A second-year *hardware/software interface* course does not, by design, use high-level languages such as Java where fundamental software issues are easier to teach (which is why introductory courses use them).

Senior-level courses such as operating systems, computer architecture, parallel programming, graphics, etc. can and should discuss concurrency. Indeed, a key point of earlier exposure is to have shared background and not have to "start threads from scratch" long after students should have seen them.

2.2 Where This Material Did Fit

Our second-year data-structures course follows introductory programming and a discrete math course (boolean and first-order logic, induction, finite-state machines, sets and relations, etc.). The parallelism and concurrency unit follows classic material on asymptotic complexity, balanced trees, priority queues, hashtables, sorting algorithms, and graphs. In our revision, we removed (or moved to senior-level algorithms) several less common data structures and algorithms. NP and NP-completeness is in another required course, but on a semester system it might fit with data structures.

The key insight is that an introduction to shared-memory multithreading, divide-and-conquer parallel algorithms, and lock-based mutual exclusion fits very well in this course.

3. What The Material Covers

Several (debatable) principles guide the presentation of the material and what is omitted despite being valuable.

1. *Distinguish parallelism and concurrency.* By parallelism, I mean using extra computational resources to solve a problem faster. By concurrency, I mean correctly and efficiently managing access to shared resources. While using these terms in this way is not entirely standard, the distinction is paramount.
2. *Stick to shared memory.* Without time for multiple programming models, mention message-passing, dataflow, and data parallelism only briefly. Shared memory fits best with the rest of the course, which ignores external storage except when studying B trees.
3. *Focus on programming patterns and guidelines.* It is not enough to explain how threads or locks work. Idioms are essential.
4. *Use Java and Java 7's ForkJoin Framework.* Present threads, lightweight tasks, locks, and condition variables as primitives. Do not discuss how to implement the primitives, schedule threads, etc. That is, stick to the application-developer's view.

* This position paper was accepted for presentation at the 2010 Workshop on Curricula for Concurrency and Parallelism, held at SPLASH 2010.

The unit begins by introducing shared memory (multiple threads with private call-stacks, but one shared heap) and some Java threading basics. We then distinguish parallelism from concurrency as defined above. We then study parallelism first, using no synchronization other than fork-join. The parallelism approach is heavily influenced by the work of Guy Blelloch, Charles Leiserson, and others. Unlike them, I am not an expert: anyone who can teach data structures can teach this material.

3.1 Parallelism

To motivate fork-join parallelism, we consider an algorithm for summing an array's elements using an explicit number of threads. We discuss reasons this algorithm has brittle performance. For example, the number of processors available to a user program may fluctuate. We therefore turn to a divide-and-conquer approach where a thread sums a subrange by summing the two halves in parallel and then adding the results. Because this style is infeasible using Java's heavyweight threads, we use (a sliver of) Java's ForkJoin Framework, for which I wrote notes suitable for beginners. We do not program in terms of *built-in* patterns like reductions over arrays because students benefit from first seeing and analyzing the efficiency of patterns like reductions done manually.

We then analyze parallel algorithms in terms of work T_1 (the time it would take 1 processor) and span T_∞ (the time it would take an infinite number of processors). We state without discussion of implementation that the framework gives an expected run-time guarantee of $O(T_1/P+T_\infty)$ for P processors and discuss why this is asymptotically optimal. We then discuss Amdahl's Law, and for homework students plot depressing consequences like how much of a program's execution must be parallelizable to achieve a 50x speed-up given 256 processors.

After concluding that many problems can be solved just with parallel maps and reduces, we learn a couple surprising non-trivial parallel algorithms. I chose parallel prefix, parallel quicksort (including using auxiliary storage to parallelize the partition), and parallel mergesort. The first two fit together well because quicksort's parallel partition uses parallel prefix as a subroutine.

3.2 Concurrency

The concurrency unit is presented later, after students are comfortable with multiple threads of execution. We point out that our fork-join programs never accessed the same memory at the same time. But suppose data structures considered earlier in the course (stacks, priority queues, hashtables, etc.) need to handle concurrent access correctly. What does that mean? How do we do it?

To stick with widely-available technology, I focus on using Java's locks for achieving mutual exclusion. The primary danger we discuss is race conditions, but here I draw another underappreciated distinction. There are *data races*, which must always be avoided, and there are *bad interleavings* (often called higher-level races), the definition of which depends on the abstraction being implemented. From this perspective the term "race condition" is annoyingly unilluminating: I wish we called data races *simultaneous access errors*.

Locks are difficult to use correctly in non-trivial programs, so a full lecture discusses programming guidelines, such as those sketched in the first chapter of Goetz et al's Java Concurrency In Practice. I emphasize that most memory should be thread-local or immutable (a great reason for functional programming), that critical sections must be neither too small nor too large, that coarse locking is an easier starting point, etc. Students get good at finding and explaining bad interleavings for short examples like implementing a stack "peek" operation as a "pop" followed by a "push."

Left until the end are important discussions on deadlock, reader/writer locks, and condition variables. Deadlock can wait because simple examples acquire one lock at a time. Reader/writer locks fit well with the issue that simultaneous reads are okay, which arises when discussing data races. Condition variables are difficult to teach to novices, particularly since the `wait` and `notify` methods in Java's `Object` class do not support two conditions attached to the same lock, which is the preferred way to implement the canonical example of a bounded buffer. It is tempting to drop condition variables, but I feel some notion of passive waiting and notification is part of a proper introduction. Perhaps a blocking queue could be presented as a "primitive" on top of which something else like a pipeline could be built.

4. Why Data Structures Is a Natural Place

By teaching the material at a high level of abstraction, it fits well with the other topics in the course. Students have just learned divide-and-conquer sequential algorithms and how to reason asymptotically, so divide-and-conquer parallel algorithms are natural. Constant-factor issues are also analogous: A particularly nice connection is teaching that in practice sequential quicksort switches to an $O(n^2)$ sort for small n, exactly like parallel algorithms switch to sequential variants below a cut-off. More generally, a major theme of a data-structures course is the power of an $O(n)$ height tree having $O(2^n)$ nodes, which is why simple divide-and-conquer algorithms have exponential parallelism.

For concurrency, data structures like queues and hashtables provide most of the canonical examples for bad interleavings. Revisiting previous abstractions and considering thread safety is fun and timely. A bounded buffer is a queue that blocks instead of raising exceptions when it is empty or full. Mostly-unchanging hashtables can motivate reader/writer locks.

5. Available Materials

Unaware of suitably short and introductory texts for this material,[1] I developed my own materials. I am very eager to have others use and adapt my work. I have available at `www.cs.washington.edu/homes/djg/teachingMaterials`:

- Written reading notes for students (and instructors!) that cover all the material, about sixty pages in total
- Lecture slides, paper-and-pencil homework exercises, and sample exam questions
- A programming project using parallelism to (at least in theory) more quickly process real U.S. census data. It comes with a simple GUI that makes the program much more fun to use.

I strived to extract the materials from the particulars of my course so that they can be easily adapted. I have also provided the source-code for everything, even the reading notes, and have no problem with any educator doing whatever they want with the materials.

6. Status and Call-For-Feedback

I taught this course for the first time in Spring 2010 and it was a great success. In Summer 2010, graduate student Tyler Robison successfully taught the course as well (he was a teaching assistant for the Spring offering). Three other faculty will teach the course over the next year. While I am sure each instructor will significantly adapt the topics and materials to his/her needs, I feel strongly that

[1] Cormen et al's Introduction to Algorithms, 3rd Ed. has a more advanced and high-level take on fork-join parallelism, but no mutual exclusion. Horstmann and Cornell's Core Java, Vol. 1, a recommended reference for my students, discusses Java's threads, as do many other programming references. A full discussion of other options is beyond our purpose here.

the material does not and must not require expertise in multithreading. Experts should develop complementary senior-level courses. I believe such courses will be much better if students "arrive" with the fundamentals described here behind them.

While I am seeking to advertise my work so that it may prove broadly useful, I am also eager to receive constructive feedback. What could be improved? What is too difficult for others to teach? How can this material fit in different curricula? How can the (rough-draft) written course notes better explain the material?

Acknowledgments: Tyler Robison and Brent Sandona were great teaching assistants for a new course. Larry Snyder was a valuable sounding board. The 2009 Multicore Programming in Education workshop participants heavily influenced my design.