Introducing Parallelism and Concurrency in the Data Structures Course

Dan Grossman Ruth E. Anderson
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA, USA
djg@cs.washington.edu rea@cs.washington.edu

ABSTRACT

We report on our experience integrating a three-week introduction to multithreading in a required data structures course for second-year computer science majors. We emphasize a distinction between parallelism and concurrency that teaches students to use extra processors effectively and enforce mutual exclusion correctly. The material fits naturally in the data structures course by having the same mix of algorithms, programming, and asymptotic analysis as the conventional, single-threaded part of the course.

Our department has used this unit for 1.5 years and will do so indefinitely. We report feedback from students, multiple instructors for the course, and students in a later course that uses threads. We developed a full set of course materials that have been adapted for use by instructors in various courses at five other institutions so far.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum*

General Terms

Algorithms, Experimentation

Keywords

Parallelism, Concurrency, Data Structures, Fork-Join, Mutual Exclusion, Undergraduate Curriculum

1. INTRODUCTION

As multiprocessors become increasingly common across the computing landscape, there is widespread interest in expanding coverage of threads, parallelism, concurrency, synchronization, etc. in undergraduate computer science curricula. However, as described in a recent exploration of the subject [3], there is not yet consensus on where in the curriculum to introduce these topics and what fundamental concepts are most important. In the near term, many institutions may find it equally unrealistic to, on the one hand, modify many courses so that multithreading pervades the curriculum or, on the other hand, add an entire required course. Instead, our approach has been to use *part* of a required lower-level course, roughly three weeks, to introduce topics like threads in such a way that various advanced courses, such as computer architecture, operating systems, databases, algorithms, graphics, and many others, can rely on them.

This paper reports on our successful experience introducing parallelism and concurrency into our department's required data structures course. Since Spring 2010, five instructors, ourselves included, have taught the course, with a range of background in parallelism from none to moderate and teaching experience from graduate student to senior faculty. Success requires no more experience than what is already needed to teach data structures – if you can learn topics like B-Trees and Dijkstra's Shortest-Path Algorithm well enough to teach them, then you can learn everything in our unit. All our materials are freely available and have already been adapted by instructors at five other institutions.

The central pedagogic theme is a clear distinction between *parallelism* − using extra computational resources to solve a problem faster − and *concurrency* − correctly and efficiently managing access to shared resources. We start with parallelism and divide-and-conquer fork-join algorithms before introducing the more difficult concurrency topics such as race conditions and deadlock. Just like in the rest of the data structures course, the material covers programming constructs and pragmatics (in Java, though language choice is not crucial), algorithms, asymptotic complexity, and constant-factor overheads. We emphasize fundamental problems like computing a reduction (e.g., a sum) over an array in parallel or ensuring an API enforces mutual exclusion. Given only three weeks, we focus on shared-memory programming, leaving message passing and distributed programming for more advanced elective courses.

Our goals in sharing our experience are to:

- Convince others that a data structures course is a very natural place in a computer science curriculum to make room for a general introduction to parallelism and concurrency. The topics rely on and reinforce core ideas already in the course.
- Elucidate parallelism versus concurrency as a core theme.
- Encourage adoption of our materials, including reading notes, slides, sample homework and exam questions, and a programming project.
- Share mostly-positive feedback from students, instructors, and students in a later course (namely operating systems).

With these goals, the rest of the paper largely describes the course-unit content and motivation (Section 3) after a brief primer on our larger curriculum context (Section 2). We then discuss why data structures is a great fit for the material (Section 4). Section 5 shares feedback from various stakeholders: students, instructors, and those who have adapted the materials at other institutions. Section 6 discusses related work. Section 7 describes the freely available materials and concludes.

The materials described in this paper are actively maintained at [10].

2. CURRICULUM CONTEXT

To put our three-week curriculum unit in context, this section briefly describes our other low-level courses in general and the data structures course in particular. However, the unit should fit well somewhere in most computer science curricula.

Our university is on a ten-week quarter system, with most computer science courses (including data structures) having three 50-minute lectures and one 50-minute recitation section per week. The data structures course ranges in size from 30 – 100 students and is a pre-requisite for most advanced courses. It comes after a two-quarter introductory programming sequence and a discrete structures course (boolean and first-order logic, induction, finite-state machines, sets and relations, undecidability, etc.). Other low-level courses, not pre-requisites for data structures, cover lower levels of abstraction (C, assembly, binary); hardware design; probability, statistics and P vs. NP; programming languages; and software design and implementation.

The data structures course is fairly conventional, building on the initial coverage of stacks, queues, and binary search trees in the introductory courses. The course covers asymptotic complexity, priority queues, balanced search trees (AVL trees, B-Trees), hashing, sorting, graphs, and graph algorithms. To make room for multithreading, we reduced coverage of amortization and removed more obscure priority-queue implementations, disjoint-sets (union/find), and network flow, some of which will move to an advanced elective algorithms course. The course has (Java) programming and paper-and-pencil homework exercises. The parallelism and concurrency unit comes late in the course.

3. CONTENT AND MOTIVATION

This section describes the core concepts covered by our materials. The unit includes 8-10 hours of instruction, two homework assignments, and a programming project.

3.1 Introduction to Multithreading

To introduce the unit, we identify the key assumption made in prior courses that is false and that we will remove: one thing happens at a time (sequential execution). Given multiple threads of execution, we can cover three main questions:

- What are some programming techniques for creating and controlling threads and letting them communicate?
- How can parallel algorithms run faster (asymptotically and/or in practice) than sequential algorithms?
- How must software be written differently so that multiple threads can access the same resources without error?

The second question above is about *parallelism*, using multiple computational resources effectively. A canonical example is using P processors to sum n numbers in time $O(n/P + \log n)$. The third question is about *concurrency*, correctly and efficiently managing access to shared resources. A canonical example is a dictionary (e.g., a hash table) that allows lookup operations from multiple threads simultaneously but ensures that insert operations occur only while all simultaneous operations wait. This distinction pervades the entire unit, and all issues of concurrency (the more difficult topic) are delayed until the second half after students are more comfortable with threads and parallel algorithms.

After introducing the basic notions of parallelism and concurrency, we then cover the *shared-memory* model: each thread has its own call-stack and control flow, but heap objects are potentially shared (though it is good style to share very few). *Communication* occurs when threads have references to the same memory, one writes, and another reads. Students have little trouble grasping this model. Correctness always requires *coordination*. For the parallelism portion, the only coordination (i.e., synchronization) primitive we use is *join*: one thread waits unless/until another thread terminates. We briefly mention other models (message-passing, dataflow, data parallelism), but, due to time constraints and a less natural fit in the course, do not revisit them.

3.2 Basic Fork-Join Parallelism

Shared-memory basics are all one needs to discuss our canonical example of how to use P threads (imagine P=4) to sum an array. A natural pseudocode algorithm is roughly:

```
int sum(int[] arr, int P) {
 int[] answers = new int[P];
 int len = arr.length;
 for(int i=0; i < P; i++)
   create a thread to sum from
   arr[i*len/P] to arr[(i+1)*len/P]
   sequentially; put result in answers[i]
 wait by joining on each of the P threads
 return the sum of the P values in answers
}
```

Implementing this solution in a language like Java requires some more code such as creating Thread objects and storing them in an array so that the waiting step can iteratively join on each thread, but the pseudocode is enough to demonstrate three key points:

- Interthread communication occurs using variables `arr` (read by the helper threads) and `answers` (written by the helper threads).
- The waiting step is essential coordination for reading the correct answers. Conversely, if we join on each thread (waiting for it to finish) right after creating it, then we get the correct answer, but the threads never run in parallel.
- If P processors are available, we can expect the algorithm to run approximately P times faster than a sequential algorithm, but there is some overhead from using threads.

More interestingly, we then argue that *this style of parallel programming has serious weaknesses that we can fix via divide-and-conquer recursion*, an approach we use for the rest of the parallelism section. These weaknesses include:

- The approach assumes we know how many processors are available, but we often do not. Other threads may already be running or the operating system may be using processors for other programs. If *P* is set to 4 but only 3 processors are available, this algorithm will likely run 1.5x slower than if *P* is set to 3. The number of processors available might even change while the algorithm is running.

- Though unlikely with a simple operation like sum, different subproblems might take different amounts of time (a load imbalance) and we will end up waiting for the slow portion while other processors sit idle.

The elegant-if-counterintuitive solution to these problems is to use recursion to create many more threads than processors, relying on the underlying scheduler to assign the threads to processors as they become available. In pseudocode:

```
int sum(int[] arr) {
  return sumRange(arr,0,arr.length);
}
int sumRange(int[] arr, int lo, int hi) {
  if(hi-lo < CUTOFF) //e.g., CUTOFF=500
    sum range sequentially and return it
  int mid = (hi+lo)/2;
  create thread to compute
      sumRange(arr,mid,hi) and store answer
  int leftAns = sumRange(arr,lo,mid);
  join on thread created above
  return leftAns + answer from other thread
}
```

This algorithm creates an implicit tree of recursive calls. Each node has two children where one child is a new thread. So after *n* levels of recursion, we have 2^n threads. We analyze this style of algorithm in Section 3.3.

While there are some programming details for using this style in Java (we spend a recitation section on them), they are less burdensome than most threaded programming. However, one should *not* use Java's built-in threads for this style of algorithm; they were not engineered for many threads doing small pieces of work. Instead, Java's ForkJoin Framework [8] was created exactly for this scenario and provides an asymptotically optimal expected-time guarantee (see Section 3.3).

The Java ForkJoin Framework is in the standard library of Java 7 (package java.util.concurrent) and is available for download for Java 6. Our materials give detailed stand-alone instructions for using the library, which is important because students using our materials need only a few methods from three classes whereas the full framework supports sophisticated industrial-strength use. No special hardware is needed, though having at least 4 cores makes parts of the project more meaningful.

Many other languages have similar frameworks (based on the pioneering work of Cilk [2]), including Intel's Cilk+ and Thread Building Blocks for C++ [6] and Microsoft's Task Parallel Library [9] for C#. In the near future, we hope to partner with instructors using languages other than Java to produce adapted versions of the materials. Such adaptation should be entirely straightforward because we use only basic features.

3.3 Analyzing Parallel Programs

Learning to use asymptotic complexity to analyze the efficiency of algorithms is a key component of our data structures course and parallelism should be treated analogously. Fortunately, the fork-join style makes this easy.

We analyze a parallel algorithm in terms of *work* and *span*. Work is the time it would take one processor to complete the task, and span is the time it would take an infinite number of processors. Span is not infinitesimal because computational dependencies must be respected and it takes *O(1)* time to create a thread. We can define a conceptual directed acyclic graph of the computation where each node is *O(1)* unit of work and each edge indicates a dependency where the source must complete before the destination can begin. Then work is the total number of nodes and span is the longest path in the graph. Analyzing our recursive array-sum algorithm for an array of size *n* reveals it has *O(n)* work and *O(*log *n)* span.

But how is that useful when we have *P* processors and *P* is neither one nor infinite? Using techniques best left to an advanced course, the fork-join framework implementation provides an expected-time guarantee of *O*(*work*/*P* + *span)*. We do teach why this guarantee is within a constant factor of optimal: ignoring caching effects, *P* processors cannot do more than *P* times better than one processor nor can they do better than an infinite number of processors.

Switching to a sequential algorithm when the recursive problem size drops below a cutoff such as 500 is a key practical technique that does not change the asymptotic complexity. In terms of the computational DAG, the array-sum algorithm produces a binary tree of threads and this cut-off trims out the bottom nine layers of the DAG, i.e., almost all the threads, with the span still logarithmic in the array size.

Finally, we study Amdahl's Law, which analyzes a program's speed-up as you add more processors under the basic assumptions that (1) part of the program parallelizes perfectly (*O(1)* span) and (2) the rest gets no benefit from more processors (inherently sequential). Amdahl's Law is three lines of simple algebra, but its negative ramifications (e.g., to get 100x speed-up from 256 processors, you must be able to parallelize more than 99% of a program's execution) are best appreciated by having students plot some curves that it implies.

3.4 More Parallel Algorithms

The array-sum example in Section 3.2 is an example of a *reduction*, which means a single result is produced from a collection using an associative (but not necessarily commutative) operator. Many computations are reductions, such as finding the maximum or finding the left-most (or right-most) element satisfying a property. One could program a reduction pattern once and for all by taking in a function object or closure for the associative operation, but we believe students benefit from first writing several reductions manually.

Even simpler than reductions are *maps*, in which a new collection is produced by applying an operation independently to each element of an input collection. Maps and reductions are the workhorses of parallel programming, so we emphasize them. Tying back into a discussion of data structures, we also note that arrays and balanced trees are more suitable for parallelization than ordinary linked lists.

But stopping here leaves the misimpression that what can be parallelized is only what is "obvious" when, in fact, just as non-obvious data structures can provide exponential speed-ups in sequential code, non-obvious parallelization techniques can provide exponential parallelism.

The core "fancy" algorithm we choose to discuss is for *parallel prefix-sum* [7]. The prefix-sum problem is to take an `int[] in` and produce an `int[] out` where `out[i]` is the sum of `in[0]..in[i]`. An *O(n)* sequential solution is suitable for a CS1 exam since `out[i]=in[i]+out[i-1]`. A surprising parallel version with *O(n)* work and *O(*log *n)* span works with two passes over the data, with the first building a clever intermediate data structure.

The prefix-sum algorithm in turn can be used as the key trick in implementing a *parallel pack* operation, in which we create an array `out` that has no empty spaces and contains exactly those elements of an array `in` satisfying some property (e.g., "greater than *x*"). Finding the elements is a trivial map operation; packing them into an array without spaces in parallel is not.

Lastly, parallel pack is the key algorithm we need for a parallel version of quicksort that has *O(n* log *n)* work and $O(\log^2 n)$ span, which requires parallelizing quicksort's partitioning step.

These "fancy" algorithms show that parallelism is not always obvious and that a key technique in algorithm design (sequential or parallel!) is to use known algorithms as subroutines. Our reading notes and lecture slides explain the algorithms in detail. We believe these algorithms are as elegant as any sequential material that may need to be excised to make room, and they take at most one lecture.

3.5 Concurrency, Mutual Exclusion, Locks

We then completely shift focus to concurrent access of shared data structures and the need for mutual exclusion, noting that the parallelism portion used algorithms where threads never tried to access the same data simultaneously. The concurrency portion of the material takes slightly more time than parallelism, but we describe it in less depth in this paper due to space constraints and since our approach is more conventional. We focus on using locks in general and Java's `synchronized` statement in particular to define critical sections that are the right size: neither too small for correctness nor too large to provide efficient concurrent access.

We focus on two distinct programming errors that are often muddled because both are called, "race conditions." *Data races* occur when two threads read/write or write/write the same object field without being ordered by synchronization. For reasons discussed at the end of the unit, data races are, except for expert use, always wrong in any program even if they seem right [1]. But preventing data races is not enough. *Bad interleavings,* also known as higher-level races, result from critical sections that are misplaced or too small for preserving application behavior. What is "bad" depends on the program. For example, if a stack *peek* operation is implemented in terms of a synchronized *pop* followed by a separately synchronized *push*, there are no data races but other threads may still see an illegal intermediate state of the stack. Having students identify bad interleavings and the errors that result is a rich source of homework problems. We use concurrent access to data structures as examples throughout this section.

3.6 Concurrency Programming Guidelines

Locks are notoriously difficult to use correctly, so teaching what locks are and why they are needed is setting up students to repeat all the common mistakes. We advocate teaching students guidelines for sticking to known-to-work concurrency idioms. Frankly, this material is a bit dry and students may not be ready to appreciate it, but we hope they will be able to refer back to it when using locks after the course. Our guidelines are not novel (see, for example, Chapter 1 of [5]), but students need to know about them. They include ideas like: share among threads as few objects as possible, mutate as few objects as possible (cf. functional programming), have a consistent locking protocol, and start with coarse-grained locking and then identify where thread contention occurs.

3.7 Remaining Concurrency Topics

Using locks to enforce mutual exclusion and avoid both data races and bad interleavings is the core skill we impart, but four remaining topics complete our introduction to concurrency:

- Deadlock and techniques for avoiding it
- Readers/writer locks to allow simultaneous read-only access
- Passive waiting for a condition to change and the canonical example of using condition variables to implement a buffer. (Warning: condition variables are surprisingly difficult to use correctly, but some notion of passive waiting is important.)
- Memory-model basics: programs with data races cannot be reasoned about in terms of possible interleavings [1]

3.8 Cross-Cutting Themes on Coverage

The list of topics in our three-week unit is carefully designed to cover only some of the basics "every computer scientist should know" from a particular perspective:

- It separates parallelism and concurrency, while emphasizing the similarities and differences by teaching them adjacently. In the real world, one often must deal with parallelism and concurrency together, but pedagogy is often best when it separates concepts that are combined and confused in practice. Parallelism is easier, so we do it first.
- It focuses entirely on the programmer's view: We *use* threads, locks, and fork-join. We do not *implement* them.
- We leave to other courses all issues related to scheduling.
- We do not cover distributed computing issues, notably latency and fault tolerance.

This focus gives a common foundation on which diverse advanced courses (O/S, distributed programming, networks, databases, graphics, computer architecture, etc.) can build, and it gives students the background to understand non-sequential programs (e.g., at summer internships).

4. THE DATA STRUCTURES FIT

Perhaps our most surprising conclusion is that the data structures course is a great fit for introducing parallelism and concurrency, since this is not (at least yet) common practice. Hopefully the previous section demonstrates our belief that our curriculum unit fits in data structures just as well as the graph algorithms unit, the sorting unit, etc. In this section, we briefly give a high-level view of why this fit works so well and then describe specific topic-level synergies with other course units.

At a high level, the data structures course teaches students to (1) reason asymptotically in terms of abstract models, (2) appreciate the interplay between theory and practice (connecting algorithms to code and analysis to execution time), (3) use well-known conceptual building blocks they would be unlikely to reinvent (e.g., trees with guaranteed balance) and that can be exponentially (in the technical sense) better than naïve approaches, and (4) to separate data structure interface from implementation and to appreciate that narrower interfaces can allow more flexible implementations. Our parallel and concurrency topics reinforce all these outcomes: (1) work, span, and Amdahl's Law are asymptotic ideas, (2) homework assignments consider both algorithms and coding details, (3) non-obvious algorithms provide exponential parallel speedup, and, (4) thread-safe APIs demand an even more careful consideration of interfaces and how operations may interfere with each other. In short, an introduction to threads benefits from the same mix of algorithms, programming models, implementation, and theoretical analysis that is often unique to the data structures course.

On a more detailed level, there are many nice connections with conventional topics from sequential data structures; we give a few examples. Students have just learned divide-and-conquer sequential algorithms and how to reason asymptotically, so divide-and-conquer parallel algorithms reinforce the same concepts. Constant-factor issues are perfectly analogous: A particularly nice connection is showing how in practice sequential quicksort implementations switch to an $O(n^2)$ sort for small n, exactly like parallel algorithms switch to sequential variants below a cutoff. Using a DAG to define work and span demonstrates another application of graphs. For concurrency, data structures like queues and hash tables provide most of the canonical examples for bad interleavings. Revisiting previous data structure 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. Hash tables that are rarely changed can motivate readers/writer locks.

5. FEEDBACK FROM STAKEHOLDERS

As of fall 2011, the course has been taught six times so far at UW by five different instructors. Over 250 students at UW have taken the course since Spring 2010. In addition, instructors at five other universities have adapted these materials for their use. We have not performed a broad evaluation or formal learning outcomes assessment, but we can report on our current experience based on feedback from students and instructors.

5.1 Students

Students taking the course were all Computer Science or Computer Engineering majors. The course was a required course, and a prerequisite for many other courses.

Student interest – Overall, students seemed to enjoy the parallelism and concurrency material. In Winter 2011, during the last week of the quarter, students were given an in-class free response survey to gauge their impressions of the course material (N=84, out of 94 enrolled). Responses were coded to identify how often parallelism and concurrency topics were mentioned. For comparison, the number of occurrences of data structures and other topics are also listed (some students mentioned multiple items in their response). In response to "What is the most **important** thing you learned in this course?" 29 students (35%) mentioned parallelism and concurrency topics, 57 students (68%) mentioned data structures topics, and 15 students mentioned other overarching topics like "Big-O" or "Everything". When asked "What is the most **interesting** thing you learned in this course?", 57 students (68%) listed parallelism and concurrency topics. Complete data for this one-time survey appears in Table 1. Overall, it seems that students consider parallelism and concurrency a substantial and interesting part of the course, on par with other core concepts like data structures and asymptotic complexity.

Table 1. Number of student responses mentioning various topics in response to the questions "*What was the most [X] thing you learned in this course?*" (N=84)

X	Parallelism/ Concurrency	Data Structures	Other	No Answer
Important	29 (35%)	57 (68%)	15 (18%)	0
Interesting	57 (68%)	34 (40%)	2 (2%)	0
Surprising	40 (48%)	33 (39%)	13 (15%)	1
Fun	26 (31%)	41 (49%)	19 (23%)	1

Student learning - As one data point on how well students learned the new material, we examined the final exam scores of students in Spring 2010 and Winter 2011. In Spring 2010 (N=32), the first quarter the course was offered, students performed slightly better on the data structures questions (85%) than on the parallelism and concurrency questions (77%). In Winter 2011 (N=94) their performance on both types of questions was comparable (87%). In both quarters, students performed slightly better on the parallelism questions than on the concurrency questions. Although this is only one data point, it concurs with the general feeling of instructors that students were able to learn the material and that the concurrency material was more challenging for students than the parallelism material.

5.2 Instructors

One of the authors developed the course materials and taught the course for the first time in Spring 2010. For the next four quarters, four different instructors (three faculty members, one graduate student) taught the course. These other instructors had experience teaching data structures, but all were unfamiliar with most or all of the parallelism and concurrency material prior to preparing for the course. All instructors indicated that their teaching evaluations were similar to those they normally received in other courses.

Usefulness of provided materials - All four instructors used the provided slides and reading notes and found them to be an important resource for students. Instructors used a combination of the provided written homework problems and made up their own. One instructor felt that the provided project (see Section 7) was too large and trimmed it down. Another instructor created his own project.

Course pace and structure - Multiple instructors found that teaching the concurrency programming guidelines content (Section 3.6) was unexciting. One instructor planned to de-emphasize this and the advanced concurrency material (Section 3.7) in the future, partially because he felt pressed for time. In the context of 10-week quarters, some instructors felt that the material was somewhat rushed, which can lead to a more superficial coverage of some advanced data structures topics. The instructors felt concurrency material was more challenging for students than parallelism, though both were important.

5.3 Students in Later Courses

One reason to teach parallelism and concurrency in an early and required course is so that later courses can assume and build on that material. We are just starting to see students in upper level courses who have taken a version of the data structures course containing the parallelism and concurrency unit. In Spring 2011, out of 38 students enrolled in operating systems, 20 of them had seen concurrency in the data structures course. As a preliminary effort to gauge the impact of teaching parallelism and concurrency early, in the last week of Spring quarter 2011, we conducted a voluntary web survey of students enrolled in operating systems. Nine of those 20 students responded to our survey. When asked if they felt that their experience with parallelism and concurrency in data structures helped them in operating systems, 4 of the 9 viewed the previous material as "1-Indispensible!", 4 viewed it as "2-Fairly useful", and 1 student did not answer (none listed it as "3-A little useful" or "4-Not very useful"). Student views on the usefulness of previous coverage of specific concurrency topics (those most relevant to operating systems) are shown in Table 2.

Table 2. Student responses to "*For each of the following topics, did seeing the material in data structures help your understanding of material in operating systems*?" (N = 9)

Topics	Incredibly useful	Fairly useful	A little useful	Not very useful	I do not remember seeing this
Threads	3	4	1	1	0
Race conditions / data races	6	2	0	0	1
Mutual exclusion / locks	4	4	0	0	1
Deadlock	3	3	2	0	1

5.4 Instructors at Other Institutions

In the last year, faculty at five other institutions have adapted our materials for their use. The materials were used/adapted for CS2 courses and as introductory materials for advanced parallelism/concurrency courses (course links are at [10]). All five instructors found the materials useful and when recently polled informed us that they will use the materials again.

6. RELATED WORK

A 2010 ITiCSE working group explored the issue of integrating parallelism into computer science curricula in great detail, providing a range of references on potential content and approaches that have been tried [3]. Our approach to fork-join parallelism based on recursive algorithms and work/span has been long advocated by parallel-programming leaders such as Charles Leisersen and Guy Blelloch. Cormen et al. [4] offers a more advanced and high-level take on fork-join parallelism (but no mutual exclusion). Similarly, our approach to concurrency and mutual exclusion is by no means new. What we have added is a self-contained pedagogy for teaching parallelism and concurrency adjacently and as a natural fit in a lower-level data structures course.

7. OBTAINING THE MATERIALS

All our materials have been separated from the particulars of our data structures course, and we actively maintain them (fixing errors, improving explanations, etc.) [10]. All materials are open and free, and we have posted all the sources so others can develop their own extracts or extensions. We have developed:

- Written reading notes for students (and instructors!) that cover all the material, about 65 pages in total. Some other institutions have used these as-is in lieu of a textbook while others have developed a subset to match their needs better.
- Lecture slides (PowerPoint), 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. A simple GUI (provided) makes the program fun to explore.

Based on our positive experiences using these materials in our data structures course over the past 1.5 years, we encourage others to consider adapting them to their own context.

8. REFERENCES

[1] Adve, S. V. & Boehm, H-J. 2010. Memory models: a case for rethinking parallel languages and hardware. *Communications of the. ACM* 53, 8 (August 2010), 90-101.

[2] Blumofe, R. D., Joerg, C.F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., & Zhou, Y. 1995. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on principles and practice of parallel programming*, 207-216. See also: http://software.intel.com/en-us/articles/intel-cilk-plus/

[3] Brown, R., Shoop, E., Adams, J., Clifton, C., Gardner, M., Haupt, M., & Hinsbeeck, P. 2010. Strategies for preparing computer science students for the multicore world. In *Proc. of the 2010 ITiCSE working group reports*, 97-115.

[4] Cormen, T., Leiserson, C., Rivest, R. & Stein, C. 2009. *Introduction to Algorithms*, 3rd Edition, The MIT Press.

[5] Goetz, B. & Peierls, T. et al. 2006. *Java Concurrency in Practice*. Addison Wesley.

[6] Intel Threading Building Blocks, http://threadingbuildingblocks.org/

[7] Ladner, R. E.; Fischer, M. J. 1980. Parallel prefix computation. *Journal of the ACM* 27, 4 (October 1980), 831–838.

[8] Lea, D. 2000. A Java fork/join framework. In *Proc.of the ACM 2000 conference on Java Grande*, 36-43. See also: http://g.oswego.edu/dl/concurrency-interest/

[9] Leijen, D., Schulte, W., & Burckhardt, S. 2009. The design of a task parallel library. In *ACM SIGPLAN conference on object oriented programming systems languages and applications*, 227-242. See also: http://msdn.microsoft.com/en-us/library/dd460717.aspx

[10] Current version of the materials: http:// www.cs.washington.edu/homes/djg/teachingMaterials