Practical Parallel and Concurrent Programming

Caitlin Sadowski
University of California
Santa Cruz, CA, USA
supertri@cs.ucsc.edu

Thomas Ball
Microsoft Research
Redmond, WA, USA
tball@microsoft.com

Judith Bishop
Microsoft Research
Redmond, WA, USA
jbishop@microsoft.com

Sebastian Burckhardt
Microsoft Research
Redmond, WA, USA
sburckha@microsoft.com

Ganesh Gopalakrishnan
University of Utah
Salt Lake City, UT, USA
ganesh@cs.utah.edu

Joseph Mayo
University of Utah
Salt Lake City, UT, USA
u0565813@utah.edu

ABSTRACT

Multicore computers are now the norm. Taking advantage of these multiple cores entails parallel and concurrent programming. There is therefore a pressing need for courses that teach effective programming on multicore architectures. We believe that such courses should emphasize high-level abstractions for performance and correctness and be supported by tools.

This paper presents a set of freely available course materials for parallel and concurrent programming, along with a testing tool for performance and correctness concerns called ALPACA (A Lovely Parallelism And Concurrency Analyzer). These course materials can be used for a comprehensive parallel and concurrent programming course, *à la carte* throughout an existing curriculum, or as starting points for graduate special topics courses. We also discuss tradeoffs we made in terms of what to include in course materials.

Categories and Subject Descriptors

H.3.2 [**Computers and Education**]: Computer and Information Science Education; D.2.8 [**Programming Techniques**]: Concurrent Programming

General Terms

Human Factors, Languages, Performance

1. INTRODUCTION

Once upon a time programmers were taught to write sequential programs, with the expectation that new hardware would make their programs perform faster. In the early 2000s, we hit a power wall [1]. Today, all major chip manufacturers have switched to producing computers that contain more than one CPU [24]. Parallel and concurrent programming has rapidly moved from a special-purpose technique to standard practice in writing scalable programs. The future is upon us, and many undergraduate courses are lagging behind.

Scheduling decisions made by an operating system or a language runtime can dramatically affect the behaviour of parallel and concurrent programs. This makes such programs difficult to reason about, and introduces a whole new set of code bugs students may encounter, such as data races or deadlocks. Students programming for multicore systems also face a new set of potential performance bottlenecks, such as lock contention or false sharing. Concurrency bugs are both insidious and prevalent [8, 14, 16]; we need to ensure that all computer science students know how to identify and avoid them.

We need to start teaching courses that prepare students for multicore architectures now, while concurrently updating the whole curriculum. Since parallel and concurrent programming was previously relevant only to the relatively small collection of students interested in high-performance computing, operating systems, or databases, instructors may need help developing course materials for this new subject. In this paper, we put forward several ideas about how to structure parallel and concurrent programming courses, while also providing a set of concrete course materials and tools.

1.1 Contributions

- We present course materials for a 16 week course in parallel and concurrent programming [2]. We begin by describing the course structure (Section 2), and then discuss the individual units (Section 3).
- We present a testing framework, ALPACA [2], which supports *deterministic* unit tests for parallel and concurrent programs, and allows students to effectively test their concurrent programs. This framework is described in Section 4.
- We discuss tradeoffs we made about what to include (Section 2). We situate these tradeoffs within related work, including comparisons with other courses (Section 5).

2. COURSE STRUCTURE

Practical Parallel and Concurrent Programming (PPCP) is a semester-long course for teaching students how to program parallel/concurrent applications, using the C# and F# languages with other .NET libraries. Taken as a whole,

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

this 16 week (8 unit) course is aimed at upper-division undergraduate students. However, individual units can also be taught *à la carte* and could be integrated into various stages of an existing curriculum. Alternatively, selections from the course materials can form the core structure for an in-depth graduate special topics course.

Parallel and Concurrent.

In the course title, we make a distinction between parallel and concurrent programming. Parallel programming is about improving performance by making good use of the underlying parallel resources of a particular machine (such as multiple cores), or the set of machines in a cluster. Concurrent programming is about improving responsiveness by reacting to simultaneously occurring events (coming from the network, user interface, other computers, or peripherals) in a timely and proper fashion. Programs may exhibit both concurrent and parallel characteristics, thus blurring the distinction; we make such a distinction to emphasize that PPCP includes both parallel and concurrent programming. Other courses may only focus on one of these two facets (Section 5).

Performance and Correctness.

In this course, students learn to write parallel algorithms, and explain (and fix) both performance and correctness bugs. PPCP has a strong emphasis on both correctness and performance, with a slight tilt towards correctness. On the performance side, students will learn how to predict and test parallel speedups, and explain actual performance bottlenecks. On the correctness side, students will develop a vocabulary for reasoning about parallelism and correctness, analyze parallel or concurrent code for correctness, and understand expected invariants in their code.

Breadth-first, Abstraction-first.

PPCP has a breadth-first structure. Since there is not a clear consensus on whether any one parallel or concurrent programming model will emerge as a clear winner [22], we feel that students should be exposed to a variety of paradigms and topics. This course is tilted towards shared memory systems with multiple threads, since we feel it is important to prepare students to succeed in this common case. However, a wide breadth of material is covered, including message passing, data parallelism and even some GPU programming.

This course emphasizes productivity and starts with a high level of abstraction. Abstraction layers, such as the .NET Task Parallel Library (TPL) [15], allow students to quickly write parallel or concurrent programs while avoiding explicit thread management. Figure 1 shows an example generic `ParQuickSort` function which uses `Parallel.Invoke` from the TPL to run quicksort recursively in parallel; programmers do not have to explicitly create or manage threads.

Unfortunately, understanding performance bottlenecks requires moving through abstraction layers. For example, discovering which portions of a program lie along the "critical path" of places where optimizations result in parallel speedup requires analyzing the (high level) task dependency graph for a program, whereas preventing false sharing requires understanding the (low level) cache behaviour. After abstractions are introduced, PPCP shows students how to break those abstractions when necessary for performance. Students learn the low-level building blocks (such as monitor synchronization and threads) that are combined to build TPL-level abstractions, and the performance effects of these building blocks interacting with a parallel architecture.

```
static void ParQuickSort<T>(T[] arr, int lo, int hi)
            where T: IComparable<T> {
  if (hi-lo <= Threshold) InsertionSort(arr, lo, hi);
  else {
    int pivot = Partition(arr, lo, hi);
    Parallel.Invoke(
      delegate {ParQuickSort(arr, lo, pivot-1);},
      delegate {ParQuickSort(arr, pivot+1, hi);}
    );
  }
}
```

Figure 1: Parallel quicksort with the TPL; Parallel.Invoke() runs a list of delegates in parallel.

Tool-based learning.

PPCP supports a tool-based approach to correctness and performance. We have developed a tool called Alpaca (A Lovely Parallelism and Concurrency Analyzer) which students can use to explore performance and correctness concepts. Students will build an understanding of correctness conditions and performance problems through experimentation. Throughout PPCP, students will regularly analyze their code, including tests of data race detection, deadlock detection and linearizability checking. These tests leverage the CHESS stateless model checker [20] to run unit tests on different thread schedules, increasing the chances that a scheduling-dependent assertion will be tripped. Alpaca tests can also link into a GUI-based performance analysis tool (called TaskoMeter). We believe that this tool-based approach will improve the learning experience by enabling quick and simple analysis to make debugging less frustrating; preliminary observations of students in our Fall 2010 pilot course support this assertion. We have also found Alpaca and TaskoMeter to be extremely useful in debugging our own code. The Alpaca framework is discussed in more depth in Section 4, and is available for download [2].

In addition to Alpaca tool support, the course materials also include slides and lecture notes, which are supported by a selection of programming samples and a recommended textbook. The programming samples include a variety of appealing examples, including small interactive games and visual examples such as ray tracing programs; additional samples are available online [18]. Further reading on course topics can be found in a recommended textbook [6], which is also available online; suggested sections from this textbook are listed at the end of the slides for each lecture and in the lecture notes.

3. UNITS

Figure 2 shows the dependency structure of the eight units. The course material starts at a high level of abstraction by introducing patterns rather than primitives (units 1-4). For example, in the DAG model of parallel computation tasks are represented as nodes and dependencies between parallel tasks are represented as directed edges. This model is presented in Unit 1 and can be used to reason about both performance and correctness in future units. In later units

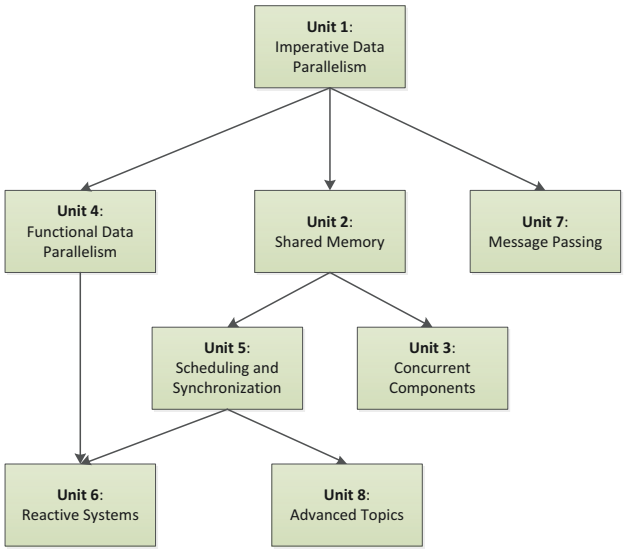

Figure 2: Unit dependencies.

(Unit 5 & 8), these abstractions are peeled back to expose the underlying primitives (e.g. threads and low-level synchronization). PPCP also includes an introduction to message passing (Unit 7) for distributed systems, and a unit specifically focused on concurrency (Unit 6).

3.1 Unit 1: Imperative Data Parallelism

Students are motivated when they initially write programs that demonstrate parallel speedups [13]. Unit 1 introduces data and task parallelism and emphasizes pleasing parallelization of independent loops. This unit teaches students to use abstractions from the TPL, such as Parallel.For loops, which execute iterations in parallel, and lightweight concurrent tasks. It also contains a set of core abstractions that can be used to reason about performance and correctness such as the DAG model of computation and Amdahl's law, which provides an upper bound on the performance gain from parallelizing a sequential program. This unit also introduces the concept of unit testing.

3.2 Unit 2: Shared Memory

Unit 2 is an in-depth look at correctness and performance issues under the shared memory model. This unit introduces the concept of locks and details strategies for preventing data races, including proper synchronization, isolation of data by partitioning across threads, and making shared data immutable. Building on these foundations, Unit 2 then teaches students to identify and fix other concurrency errors such as atomicity violations or deadlocks. This unit also delves into common performance bottlenecks, such as false sharing, data locality, task creation overheads, and synchronization overheads. The last lecture in this unit introduces a selection of high-level design patterns which can be used to avoid concurrency bugs. For example, the producer-consumer pattern reduces data sharing to a thread-safe buffer.

3.3 Unit 3: Concurrent Components

Unit 3 is focused on building thread-safe components, an important skill for many programmers [10]. This unit teaches students how to leverage interface-level specifications and determinism analysis when adding internal parallelism to a component. This unit also discusses important thread-safety concepts (such as linearizability) for reasoning about how concurrent accesses to the same component interact.

3.4 Unit 4: Functional Data Parallelism

Unit 4 emphasizes the nice match between sets and parallelism through mediums such as parallel queries with PLINQ (Parallel Language-Integrated Queries) [17], parallel functional programming with F#, and array parallel algorithms with Accelerator [25]. In the PLINQ section of this unit, students will learn how to write LINQ queries, make them parallel, and get speedup in their programs. LINQ queries provide a distinctive way to structure code and get information from data structures with database-like queries. This data-first view may help clarify the parallelism in an algorithm. Parallel functional programming with F# is an immutability-first programming style which strives to minimize global variables and so prevent errors, such as data races, caused by inadequately protected shared data. Accelerator allows students to harness the power of Graphics Processing Units (GPUs) to quickly run data parallel computations over arrays.

3.5 Unit 5: Scheduling and Synchronization

Unit 5 peels back the abstractions introduced in other units. The first part of this unit is focused on how threads are mapped to cores, and how to synchronize between threads. This unit also covers how high-level synchronization (such as semaphores) can be built from low-level synchronization primitives (such as Interlocked operations). After this, Unit 5 includes a section devoted to correctness issues for the threading model. Students learn to reason about programs as state machines and to build and analyze synchronization protocols. The last section of this unit focuses on how tasks are mapped to threads. This unit concludes with a discussion of the threadpool and scheduling decisions made by the TPL, and strategies for customizing and building a work-stealing scheduler.

3.6 Unit 6: Interactive/Reactive Systems

Unit 6 focuses on reactive programming, where a program must handle many requests from the environment in a responsive fashion. This unit focuses on various patterns for asynchronous programming, building on principles of state machines established in Unit 5. State machines underlie many of the technologies discussed, such as F#'s `async` construct and Reactive Extensions for .NET.

3.7 Unit 7: Message Passing

Unit 7 presents a tutorial on conventional MPI-style message passing programming. We wrote a simplified implementation of MPI [11] inside the C# framework so that students have a more seamless entry to this paradigm and a consistent programming language syntax structure to work with. This unit introduces students to write programs suitable for distributed systems. It also provides a simple formal model of message passing.

3.8 Unit 8: Advanced Topics

Unit 8 is a catchall collection of some advanced topics that can be included as time permits. These topics can also

```
Object[] arr;

public int ProcessArray()
{
  int accum = 0;
  for (int i = 0; i < arr.Length; ++i)
  {
    if (arr[i] != null)
    {
      accum += arr[i].count;
      arr[i] = null;
    }
  }
}
```

Figure 3: Code snippet involving a data race.

```
[UnitTestMethod]
public void ProcessTest()
{
  Console.WriteLine("Processing....");
  Parallel.Invoke(
      () => {ProcessArray();},
      () => {ProcessArray();}
  );
}
```

Figure 4: Sample unit test of the snippet in Figure 3.

be focused on in more depth for a graduate special topics course. Advanced topics currently include a section on memory models, a section that has a detailed focus on caching behaviours, a section on safe strategies for lock-free data structure creation, plus a section on alternative programming models to synchronization such as transactions or revisions [5]. We may add additional topics to this unit, based on feedback from the initial iterations of the course.

4. ATTRIBUTE-BASED TESTING

Concurrency bugs are insidious. They only manifest on particular schedules, and so cannot be found with traditional testing methods. For example, imagine there are multiple simultaneous calls to `ProcessArray()` in the code snippet in Figure 3. These calls may successfully execute hundreds of times before the program crashes. However, it is possible for two threads to simultaneously verify that a particular array entry is non-null, and then one thread tries to access the `count` field after the other thread has nulled out the entry! Testing this method may or may not cause the program to crash with a null pointer exception. Even when an error manifests, it may be extremely difficult to reproduce.

In order to effectively test parallel or concurrent programs, the test suite must somehow control the scheduling of operations, and drive execution down the path of multiple different schedules. This process is called *stateless model checking*.[1] Of course, enumerating all possible schedules would be too slow. CHESS [20] is a stateless model checking framework which uses sophisticated heuristics [19] to decide which schedules to prioritize. In addition, CHESS has built-in analyses which can be used to detect concurrency bugs such as data races or deadlocks.

The Alpaca tool allows programmers to add attributes to unit test methods that enable different types of concurrency error detection and utilizes CHESS to both run the test methods on different schedules and analyze the tests for concurrency errors. Figure 4 shows an example Alpaca test. The `[UnitTestMethod]` annotation tells Alpaca to run the `ProcessTest()` method as a unit test, without any additional concurrency checking.

[1]"Stateless" refers to the fact that there is not a separate explicit model of the program.

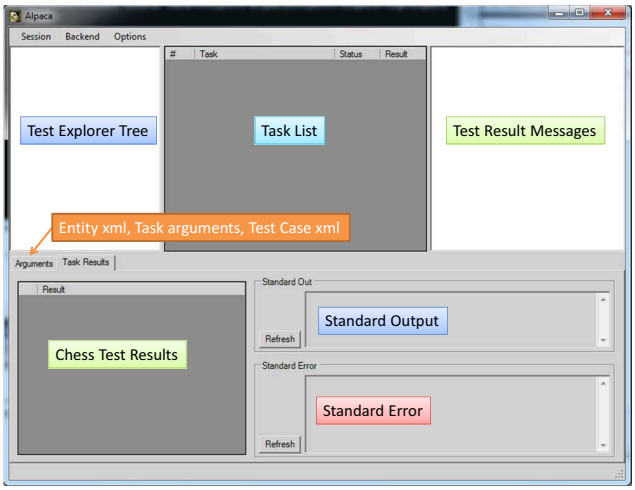

Figure 5: Screenshot of the Alpaca tool

Once the student annotates test methods with different attributes, the concurrent unit tests can be run in the Alpaca UI (Figure 5). The tree of current test methods is displayed in the top left corner; students can use this tree to select which tests to run and then run them. The task list pane (top center) displays the status of currently executing tests. The test result message pane (top right) displays the result of running each test. More detailed information about the errors found in a particular selected test will display in the chess test results pane (bottom left). The bottom right pane displays standard output or standard error. The performance overhead of logging complete information about thread interactions ("tracing") is too high to be always executing. If a test results in an error state, the student can choose to replay the test with tracing and then interactively explore the thread interactions which led to the bug.

We created Alpaca as a way to make the CHESS research tool suitable for pedagogical use. The research tool originally matched tests to executables in a one-to-one way; this meant that the programmer was responsible for creating a separate executable for each unit test. Now programmers can just annotate methods for testing. Furthermore, the research tool user needed to understand many command-line flags, and set up an XML file for the test suite. Instead of command-line flags, the new framework allows the programmer to specify different test types. We wanted to make the interface completely interoperable for different types of test methods, so we had to change the framework to provide cleaner abstractions, and support multiple test types (such as performance or unit tests).

The Alpaca framework supports several different types of attributes; we briefly overview four of them here:

- `[UnitTestMethod]`
  This attribute tells Alpaca to just run the test, without controlling the scheduling or detecting any extra concurrency errors. This attribute is similar to Visual Studio's `[TestMethod]` attribute.

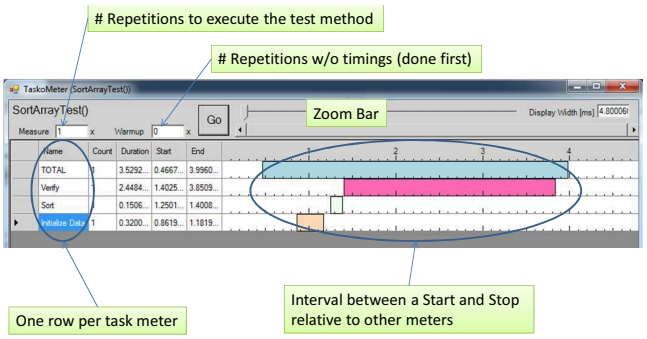

Figure 6: Screenshot of the TaskoMeter tool

- `[PerformanceTestMethod]`
  This attribute will open up the TaskoMeter user interface (Figure 6) upon running the test. The programmer must demarcate code sections where performance measurements are desired. TaskoMeter displays timing information across different threads within these programmer-defined sections.
- `[ScheduleTestMethod]`
  When a test with this attribute is run, it will run under the CHESS direct execution model checker, to test for deadlocks and assertion violations on different schedules.
- `[DataRaceTestMethod]`
  A test with this attribute behaves very similar to one with the `[ScheduleTestMethod]` attribute. However, the system explores less schedules but checks for data races on those observed schedules.

5. RELATED WORK

5.1 Unit Testing in Education

A number of researchers have advocated the inclusion of test-driven development into the computer science (CS) curriculum [12, 21, 23]. However, testing concurrent programs is extremely difficult [14]. Because testing frameworks usually do not control scheduling of multiple threads, the results of testing are nondeterministic. In particular, tests do not necessarily expose concurrency-specific bugs such as data races or deadlocks (Figure 3).

Previous pedagogical tool suites exist [3, 7]. However, Alpaca is unique in being tightly coupled with both a set of course materials and a unit testing framework. Ricken et al. [21] present some examples of concurrent unit tests in ConcJUnit; these examples are suitable for an educational context. ConcJUnit is a framework that ensures that all threads spawned in unit tests terminate, but does not explicitly check for concurrency-specific bugs (such as race conditions). Furthermore, since ConcJUnit tests do not control scheduling, they may be nondeterministic. In contrast, Alpaca tests check for concurrency-specific errors, control scheduler-dependent non-determinism, and run the unit tests on different thread schedules.

5.2 Multicore Curricula

Some researchers have advocated sprinkling parallelism and concurrency throughout the CS curriculum and introducing these topics in introductory courses [9, 4]. Although we believe that concurrency should be better integrated with the entire CS curriculum, doing so requires widespread changes. Additionally, special-purpose parallel and concurrent programming courses can help fill the gaps for current students while introduction of concurrency in introductory courses is phased in. For these reasons, the focus of this paper is on the content and structure for a dedicated parallel and concurrent programming course. However, individual course units could be used *à la carte* at different points throughout a curriculum. Additionally, the Alpaca framework could be used independently from the course materials.

Suzanne Rivore [22] presents a breadth-first course in multicore programming. This course is an upper-division elective which covers three different paradigms: shared memory with OpenMP, pattern-based parallel programming with Intel's Thread Building Blocks (TBB), and a graphics processors (GPU) programming model with nVidia's CUDA library. These three paradigm-focused course subsections were followed by reading technical papers on other advanced topics within parallel and concurrent programming. Performance profiling was also an important part of the code development process for students. Students found that learning the different syntax and style of the three paradigms was challenging. Although PPCP covers a wide breadth of paradigms, these paradigms are presented in an interoperable framework. We discuss shared memory with threads and synchronization in Unit 5, pattern-based parallel programming with the TPL in Unit 1, and GPU programming with Accelerator in Unit 4: all of these programming models use C# syntax and the same development environment.

5.3 Comparison to Other Courses

The University of California at Berkeley and the University of Illinois at Urbana-Champagne both make up the well-funded Universal Parallel Computing Research Center (UPCRC). For this reason, we focus the following section on courses taught at these schools.

GPU-based courses.

Several courses exist which target programming on graphics processing units (GPUs). These include "Applied Parallel Programming" at UIUC, which is focused on CUDA programming tools and scientific computing workloads. We briefly touch on GPU programming with the presentation of Accelerator in Unit 4. However, PPCP is focused on general programming and has more breadth.

Multicore Programming Courses.

Several multicore-themed courses exist, including Berkeley's "Parallel Programming for Multicore" and "Applications of Parallel Computers". These courses are mostly focused on shared memory programming using Posix threads. "Parallel Programming" at UIUC starts with several lectures on MPI, and then moves on to object-based parallel programming (Charm++), followed by OpenMP and a few other language models (such as high performance FORTRAN) and performance bottlenecks.

All the above courses are more focused on parallelism than concurrency and performance over correctness. Also, they do not include the tool support found with Alpaca. Unlike these courses, PPCP leverages the TPL to start at a high level of abstraction, and provides a unified framework for introducing a variety of programming models.

6. FUTURE WORK

A pilot version of PPCP is being taught at the University of Utah in Fall 2010, while this article goes to press. A second pilot will be taught at the University of Washington in Spring 2011. We are collecting survey data from the students enrolled in the initial pilot course; we also have developed feedback surveys for teachers who use PPCP materials and for anyone who reviews the online course materials. We plan to incorporate feedback from these evaluations.

We are working to develop additional materials; we are currently writing more example programs, as well as slides for lab sessions.We are also incorporating new tests into ALPACA, such as MPI analyses.

7. ACKNOWLEDGEMENTS

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

8. ADDITIONAL AUTHORS

Additional Authors (Redmond, WA, USA): Madanlal Musuvathi (Microsoft Research, email: madanm@microsoft.com), Shaz Qadeer (Microsoft Research, email: qadeer@microsoft.com), and Stephen Toub (Microsoft, email: stoub@microsoft.com)

9. REFERENCES

[1] 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.

[2] T. Ball, S. Burckhardt, G. Gopalakrishnan, J. Mayo, M. Musuvathi, S. Qadeer, and C. Sadowski. Practical parallel and concurrent programming course materials. `http://ppcp.codeplex.com/`.

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

[4] K. Bruce, A. Danyluk, and T. Murtagh. Introducing concurrency in CS 1. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2010.

[5] S. Burckhardt, A. Baldassion, and D. Leijen. Concurrent programming with revisions and isolation types. In *Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

[6] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2010. `http://parallelpatterns.codeplex.com/`.

[7] 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.

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

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

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

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*. The MIT Press, 1999.

[12] D. Janzen and H. Saiedian. Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2006.

[13] 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.

[14] E. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Notices*, 44(10):227–242, 2009. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[16] 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.

[17] Microsoft. Parallel Language-Integrated Queries (PLINQ). `http://msdn.microsoft.com/en-us/library/dd460688.aspx`.

[18] Microsoft. Parallel programming samples for .NET 4. `http://code.msdn.microsoft.com/ParExtSamples`.

[19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Notices*, 42(6):446–455, 2007.

[20] 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.

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

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

[23] J. Spacco and W. Pugh. Helping students appreciate test-driven development (TDD). In *Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[24] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):16–20, 2005.

[25] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Operating Systems Review*, 40(5):325–335, 2006. `http://research.microsoft.com/en-us/projects/Accelerator/`.