Parallel Programming: Design of an Overview Class

Christoph von Praun
University of Applied Sciences Nuremberg, Germany
praun@acm.org

Abstract

We designed an introductory parallel programming course at the bachelor level. The class differs from other courses in its structure: The course is organized along the *tiers of parallelism* [25]. The tiers categorize abstractions and concepts that a software developer can choose when crafting a parallel program. The tiers are from higher to lower abstraction levels: (1) automatic/implicit parallelism, e.g., parallel libraries; (2) deterministic parallelism, e.g., at the level of independent loops; (3) explicitly synchronized, e.g., shared memory with locks; (4) low-level concurrent programming with data races, e.g., lock-free data structures. The goal of the class is to introduce fundamental principles of parallel systems and to expose students to all tiers in the architecture of a parallel system. The course serves as a platform for further exploration in specialized classes.

The course has a significant share of lab sessions and programming projects. We chose the programming language X10 as the core technology and found that it facilitates the learning and rapid application of concepts at different abstraction layers and programming models. The language permits to specify common forms of parallelism, data sharing, distribution, and synchronization with succinct syntax and support for an eclipse-based IDE. We report on our first experience in teaching this course, which resulted in very positive student feedback.

Categories and Subject Descriptors K.3 Computers and Education [*K.3.2 Computer and Information Science Education*]: Curriculum

General Terms Design, Experimentation, Measurement, Performance, Theory

Keywords Concurrency, Parallelism, Curricula

1. Introduction

The ubiquity of multicore processor architectures and the additional complexity in developing efficient software for those architectures suggests that students should be trained specifically to master the upcoming challenges. Industry leaders and researchers have thus demanded that parallel programming should be in undergraduate curricula [12].

A common proposal is to integrate aspects of concurrency and parallelism across several classes in computer science and software engineering curricula [22, 26]. The key guiding principle of this approach is to 'learn what you need when you need it' [25]. At least two factors make the implementation of an integrated curriculum challenging: First, there is a natural inertia in evolving the overall curriculum and reaching consensus among faculty members [1]. Second, unlike mature fields like compiler construction or automata theory, there seems to be no established method on how to introduce students to concurrency and parallel programming. Most courses on the subject expose students to several programming paradigms by means of examples to cover breadth [16, 23]. Participants of the Workshop on Integrating Parallelism Throughout the Undergraduate Computing Curriculum (CPATH) associated with PPoPP 2011 found that "more research needs to be done" before it is possible to reach consensus and establish a common method of teaching concurrency. That said, the participants also agreed that given the current trend toward multicore computing, some exposure to concurrency early in the curriculum is urgently needed.

Since it will be a while until a common methodology of teaching concurrency is established and integrated in undergraduate curricula, we present a pragmatic approach, namely an 'orientation' class for concurrent programming at the bachelor level. The class provides students with the foundations of the field, i.e., key insights that we expect will not change in the the foreseeable future, and further exposes students to the different abstraction levels at which parallel programming can be done. The class is designed to be the starting point of further specialized courses with focus on specific programming models or applications domains.

Many courses on parallel programming approach the field from the computer architecture [13] or the scientific computing [17] area. This class puts emphasis on software architecture aspects of parallel systems. The course is organized along the *tiers of parallelism* [25]. The tiers categorize abstractions and concepts that a software developer can choose when crafting a parallel program and reflect the layered software architectures commonly found in parallel systems. The tiers are described in detail in Section 2.

Contributions

- We present the idea of structuring an introductory parallel programming class along different abstraction layers found in a parallel system.
- We report our experience with this course structure and share the teaching materials online at http://www.in.ohm-hochschule.de/professors/praun/pp/.

2. Tiers of parallelism

We adopted the idea to categorize topics that arise in the field of concurrent and parallel systems into *tiers of parallelism* [25] from Scott; since layered architecture are common in computer systems, this structure and categorization may well have appeared earlier.

Figure 1 shows the abstraction hierarchy.

ACM SIGPLAN X10 Workshop June 4, 2011, San Jose.
Copyright © 2011 ACM [to be supplied]. . . $10.00

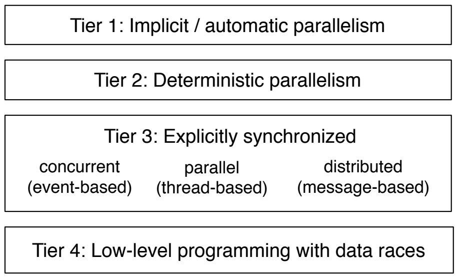

Tier 1: Implicit / automatic parallelism
Tier 2: Deterministic parallelism
Tier 3: Explicitly synchronized
concurrent (event-based)
parallel (thread-based)
distributed (message-based)
Tier 4: Low-level programming with data races

Figure 1. Tiers of parallelism [25]

We characterize each layer by examples of programming techniques and abstractions found at the respective layer. Notice that some programming languages and models may be attributed to different layers, depending on which features a programmer uses.

As a general rule, the *expressiveness* of abstractions increases as the programmer moves towards lower layers. This means that programs at higher layers can be expressed in terms of abstractions found at a lower layer but not the other way around.

The idea to structure the course along these layers is adopted from classical compiler construction courses which are organized along compilation phases, moving from higher (source code) to lower abstraction levels (assembly language). Similarly, the transitioning of an algorithm to an efficient parallel implementation occurs conceptually in a stepwise – though commonly not automated – process, along which programmers consider details at different abstraction layers and levels of detail.

2.1 Tier 1: Automatic and implicit parallelism

At this most abstract layer, the programmer follows a more or less sequential programming model, i.e., parallelism is not directly exposed. There are three common techniques to achieve this.

Autoparallelization Autoparallelization transforms a sequential program into a parallel one.

Parallel kernels The programmer uses parallel implementations of computational kernels from a library. This strategy is common for linear algebra applications, e.g. LAPACK [2]. Here, the complexity of the parallel implementation is encapsulated in a library module.

Parallel frameworks The parallelization can be hidden behind a framework API and the programmer merely supplies sequential kernels to configure the behavior of an application built upon the framework. The framework implicitly defines the architecture and overall organization of the computation and data communication. Possibly complex and optimized implementation of multithreading, scheduling, communication, and synchronization are reused as parts of the framework. Examples for this technique are map-reduce [9] and web application frameworks, e.g., Struts [3].

Most programmers who harness parallel resources operate at this tier.

2.2 Tier 2: Deterministic parallelism

At this layer, parallelism can be harnessed only in the following cases:

Independent computations . If computations do not have data or control dependences they naturally can be executed concurrently, e.g., the individual iterations of a forall loop.

Deterministic idioms Despite dependences, we may consider some computations, if executed concurrently, still as deterministic [11]. Such idioms are e.g., reduction or scan, where associativity and commutativity of the base operation lead to a well-defined result, irrespective the execution order of the computations.

It is the task of the programmer to phrase a computation such that independence of computations or deterministic idioms can be asserted.

Common techniques that fall into this layer are data parallel array languages [8], the STAPL parallel container framework [27], Intel's Concurrent Collections (CnC) parallel programming model [19], and Hierarchically Tiled Arrays (HTA) [5].

Programming techniques at this layer are attractive, since their models are not tied to a specific application, yet the expressiveness and the application domains that can be tackled with a specific technique at this layer are narrow. The development of more general, highly expressive, albeit efficient programming models and systems at this layer is still subject to current research [6, 10].

2.3 Tier 3: Explicitly synchronized

This layer is more general than tier 2 in that concurrent computations are permitted to have dependencies, e.g., through common shared mutable data. It is required that a programmer identifies all such possible dependencies and augments the program using synchronization or coordination mechanisms to guard against undesired interaction among concurrent computations. Three principal programming models exist at this layer: (1) Thread-parallel systems with shared memory and (2) event-based systems, where the occurrence of events initiates concurrent execution of event-handlers. In both programming models, common synchronization and coordination concepts are mutual exclusion and condition variables. (3) In message-based systems, concurrent computations interact through send, receive, and collective operations.

All three principal models have in common, that programmers reason about the behaviors in terms of sequentially consistent interleavings of concurrent computations. This reasoning is highly complex in comparison to programs at tier 2 that behave strictly according to a sequential program logic.

Many parallel programming classes, e.g. [16, 23], focus on this layer in the abstraction hierarchy.

2.4 Tier 4: Low-level programming with data races

This layer offers abstractions that closely follow todays shared memory multiprocessor architectures. The primitive operations are atomic load, store, and compare-and-swap. In practice, reasoning about possible interactions requires a deep understanding of the shared memory model - which usually permits many more behaviors than sequential consistency. For teaching, simplifying assumptions about the ordering of operations are commonly made, e.g., when introducing Dekker's mutual exclusion algorithm or simple implementations of non-blocking data structures.

Today, very few programmers operate at this tier. Yet many concurrency classes do (still) have focus on abstractions and reasoning within this tier.

2.5 Discussion

Our experience is that constructing a parallel program and validating its *correctness* becomes more difficult at lower layers. The *performance* argument is less clear: Sometimes, it is necessary that a programmer descends in the hierarchy of layers to achieve high-performance and scalability. For example the sharing of data may be required from a performance perspective but could force the programmer into a programming model with explicit synchronization. But descending the layers may not be a performance win in all cases. Consider, e.g., highly tuned implementations of map reduce;

a programmer who decides to craft her own implementation may rarely be able to surpass the performance of efficient implementations 'canned' into libraries.

We believe that most programmers who develop parallel programs should use abstraction of higher rather than lower layers. We believe this is true already today, considering the number of web application developers vs. the number of programmers engaged in the design of lock-free data structures. But the situation is still unsatisfactory, i.e., probably more parallel programming activity should be moved to higher abstraction tiers. We discuss the two major reasons for this situation in the following paragraphs.

First, there is need to develop efficient high-level programming models for important application domains and system architectures; significant research efforts aim in this direction but the results are often not mature enough for mainstream commercial use or teaching.

Second, the focus of many classes on parallel programming are on lower, rather that higher tiers. Many introductory classes still teach Dekker style coordination (tier 4) early on; somewhat better are classes that teach explicitly synchronized programming (tier 3) in breadth on different exemplary technologies such as OpenMP, MPI, and pthreads. The situation is unsatisfactory. For teachers who do not solely base their class on principles and theory, the following dilemma arises: Either mature and well established technologies are used for teaching and running lab sessions; these technologies are mostly found on the lower tiers in the abstraction hierarchy. Or, teachers experiment with more recent proposals for deterministic parallelism and parallel frameworks - technologies that are still subject to active research and have not proved their longevity yet.

The course we designed tries to strike a balance and resolve the dilemma: We introduce students consciously to all tiers and different programming models using a single programming technology, namely the programming language X10. To repeat what we mentioned earlier: The class is designed as an orientation in which students learn the architecture of parallel systems, acquire partial knowledge and skill of programming models at different architectural layers, and deepen issues at later stages of the curriculum.

3. Language X10

While the primary intention of the course is to teach principles, we argue that the choice of programming language is still important. In particular, a programming language should permit to express simple forms of concurrency and synchronization with simple syntax – C and Java do not meet this requirement.

We chose the programming language X10 [24] as a core technology for presenting concepts for the following reasons: A single technology facilitates the learning and rapid application of concepts at different abstraction layers and programming models. The language permits to specify common forms of parallelism, data sharing, distribution and synchronization with succinct syntax. Moreover, the managed runtime and X10 specific extension of the eclipse IDE facilitate development and debugging.

4. Course structure

The following sections present the units of the course in the order in which they are presented in class.

4.1 Motivation

We introduce the fundamental shift toward multiprocessor architectures and sketch the factors that drive this trend [4]. On a simple scenario, the benefits that throughput-oriented highly parallel computing can have on the energy efficiency are demonstrated. At the same time, we emphasize the additional complexity that parallel computing incurs and motivate the theme of the course.

4.2 Principles

This section starts with a simple model of sequential program executions with data- and control dependences. The model is extended to executions of concurrent programs, where atomicity, partial ordering, and race conditions are introduced. We emphasize that the model is language independent and also that sequencing commonly specified in imperative programming languages is often unnecessary. We introduce also performance fundamentals like speedup, weak and strong scaling, and limits characterized by Amdahl's and Gustafson's Law.

Lab sessions This section is accompanied by paper and pencil exercises with 'back of an envelope' calculations of energy and scalability.

4.3 Tier 4

Like some compiler courses start out with a brief intro to a machine model and code generation [14, 15], we commence with a brief introduction to the lowest layer in the tiers of parallelism. We give an introduction to shared memory and define the behavior of simple concurrent program executions with sequential consistency. Students should experience the difficulty of reasoning about thread interleavings and obtain a sense of that fact that real machines implement even weaker models of ordering - without learning the details of weak memory models. This section serves as motivation for less complex programming models found at higher tiers.

Lab sessions The purpose of this lab session is to let students experience the harsh realities of low-level multiprocessor programming. In the programming exercise, we let students experiment with simple programs like counters or reductions that exhibit common low-level concurrency bugs: race conditions and associative nondeterminacy. We provide the skeletons and boilerplate code for main program, concurrency, and detailed instructions on how to modify and experiment with the programs.

The behavior of X10 programs with data races is currently not well defined, since the language report [24] does not specify a detailed memory model. We use the programming language Java for programs with data races - mostly to demonstrate non sequentially consistent 'surprising' behavior. Since many students are familiar with Java, the side-by-side comparison of X10 and Java skeletons supports the transition to the new programming language (X10) which is used further throughout the class.

4.4 Tier 1

Conceptually, the abstractions and their use at this tier should be as straightforward to use as abstractions for sequential programming.

We briefly present the idea of parallelizing compilers and demonstrate the limits and challenges of the technology on a few parallel loops.

The real focus in this part of the course is on programming with parallel frameworks. As an example, we present the high-level interface of a map-reduce system [9]. We emphasize the generality of this interface and its underlaying architecture, its language- and technology independence. Although not done in the course, we plan to work in the future with a map-reduce web interface designed for teaching parallel programming [7].

Lab sessions We provide students with a simple shared memory implementation of map-reduce in X10. The purpose of this lab session is not to study the internals of the framework but to develop applications like search, word count, word frequencies, etc. Note that we ask students to implement a map-reduce framework with the same interface in a later lab session associated with tier 2.

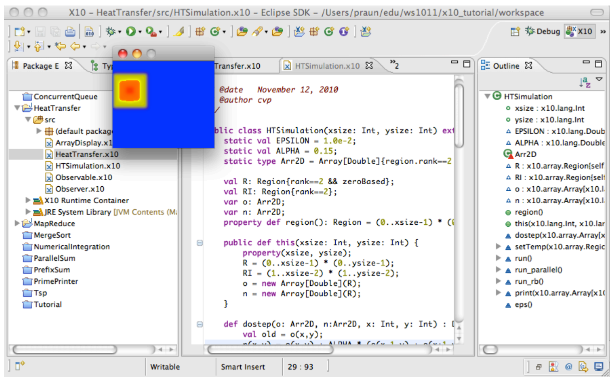
4.5 Tier 2

In this tier, we mostly worked with data and task parallel patterns [21] and exemplary application kernels, mostly from the scientific computing domain. An important objective of this part of the class is to make students realize that algorithms and data structures used in sequential computations are often not amenable to parallelization. At this tier, parallel computations are required to be fully independent. Hence, parallel decomposition goes hand in hand with the decoupling of computations by recasting the data organization and often also the algorithm of an application.

Examples for data parallelism presented in class: numeric integration, matrix multiply, and heat transfer; examples for recursive data parallelism: reduction and prefix sum.

For task parallel applications, we discuss different task scheduling strategies and their effects on performance. Example for task parallelism discussed in class: merge-sort, map-reduce framework (internals).

Lab sessions All kernels presented in class are implemented in the lab sessions based on code skeletons. For heat transfer, we include also the code for a visual display illustrated in Figure 2. The visual effect nicely demonstrated the speedups achieved on the four core processor architecture used during the lab sessions.



```
X10 – HeatTransfer/src/HTSimulation.x10 – Eclipse SDK – /Users/praun/edu/ws1011/x10_tutorial/workspace
```

```
 @date   November 12, 2010
 @author cvp
/

blic class HTSimulation(xsize: Int, ysize: Int) ext
  static val EPSILON = 1.0e-2;
  static val ALPHA = 0.15;
  static type Arr2D = Array[Double]{region.rank==2

  val R: Region{rank==2 && zeroBased};
  val RI: Region{rank==2};
  var o: Arr2D;
  var n: Arr2D;
  property def region(): Region = (0..xsize-1) * (

  public def this(xsize: Int, ysize: Int) {
      property(xsize, ysize);
      R = (0..xsize-1) * (0..ysize-1);
      RI = (1..xsize-2) * (1..ysize-2);
      o = new Array[Double](R);
      n = new Array[Double](R);
  }

  def dostep(o: Arr2D, n:Arr2D, x: Int, y: Int) : 
      val old = o(x,y);
```

Figure 2. Code skeleton and visualization of heat transfer in the eclipse IDE.

Beyond the data parallel applications, we let students study or implement (optional) the internals of the map reduce framework (120 lines of X10 code). This is an exemplary code that illustrates the full expressiveness that applications at tier 2 can achieve.

4.6 Tier 3

This section demonstrates cases, where data communication among concurrent computations is *necessary*. We clearly state that programming at tier 2 (no communication among concurrent tasks) is preferable. Necessity to communicate should arise only due to one of the following: Either performance mandates use of shared mutable data structures in memory (e.g. a shared counter), or the nature of the problem and algorithm are such that a parallel decomposition into fully independent data and computations (as taught in the section on tier 2) is not possible. As a showcase, we use the common producer-consumer problem on which we introduce the concept of critical sections (in X10: atomic blocks) and conditional synchronization (in X10: conditional atomic blocks).

Lab sessions In this lab session, students implement a producer-consumer scenario on an array-based shared queue. A simple sequential queue implementation, and the skeleton for concurrent producer and consumer computations are the starting point for exploration. The first task is to control harmful interference on the queue through mutual exclusion. Subsequently, students experiment with coordination mechanisms among producer and consumer.

Finally, we showcase a slight variation of the array queue due to Lamport [18, 20], which is lock-free but also limited in concurrency (only one producer and consumer task). While still easy to understand, the example serves to close the loop in this class to algorithms at tier 4 and is at the same time a motivation for further exploration in a subsequent class on lock-free data structures (Section 4.7).

4.7 Subsequent courses

Another class in the bachelor program is focussed on GPGPU, in particular CUDA programming.

At the master level, a course on multiprocessor programming is offered. The course is motivated by the fact that multicore computers with shared memory are the most common hardware platform that most students will encounter as young professionals. The course is structured according to the textbook of Herlihy and Shavit [18]: First principles and correctness, then performance aspects, non-blocking data structures, and finally transactional memory.

5. Experience

We report our experience with a 14-week (3 full hours per week) class held during the summer 2010. Within the undergraduate curriculum, the course is positioned as an elective class for students during their second or third year of study. All students were trained with object-oriented programming, most of the students had previously attended classes on algorithms and data structures, operating systems, or databases.

5.1 Student feedback

21 students attended class, of which 16 participated in the evaluation. The feedback was collected at the end of the course; we present the results of the evaluation as far as they relate to the theme of this article.

Our first question relates to the structure (tiers of parallelism) of the course. Figure 3 shows that most students clearly recognized the structure and found it helpful to approach the subject.

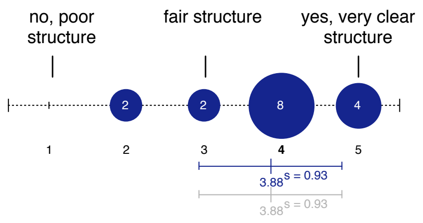


Figure 3. Has this course been well structured and did the structure support your learning?

Since this class addresses a wide spectrum of topics at different tiers, one possible concern could be that students were overwhelmed by the flood of information. Figure 4 shows that this was not the case.

One could argue that the students acquired only a partial understanding of a variety of topics and not solid skills in any of the topics. In part this is intended by the design of the class. In the final exam, we did however require students to select one focus area in which a deeper understanding of the subject was tested. It turned

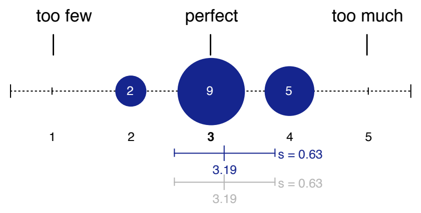


Figure 4. The number of topics and the volume of material presented in class was ...

out that the large majority of students passed this test with good or very good results.

One interesting take away from the class is that most students gave very positive feedback about the practical work done in the lab sections. Despite the workload, students liked this hands-on programming and elaborated stepwise the skeleton programs prepared by the instructor during the lab sessions and as assigned homework. Figure 5 presents feedback on the lab sessions.

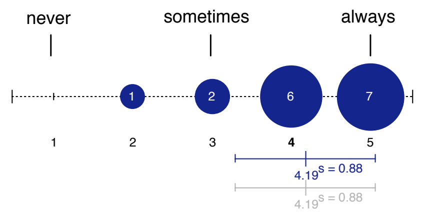


Figure 5. Did the lab sessions help you to learn and understand the materials presented in class?

The overall valuation of the course turned out very positive; the results are reported by Figure 6.

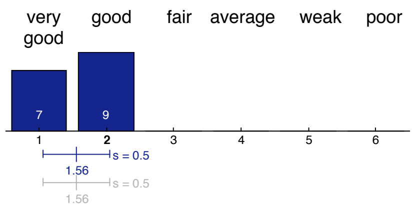


Figure 6. Students' overall impression of the class.

5.2 The role of X10

About one third of the students found the transition from Java respectively C# to X10 difficult and would have wished an X10 language tutorial up front. The author provides such tutorial online at http://www.in.ohm-hochschule.de/campus/professors/praun/x10tut/. Alternatively, students were allowed to use other programming languages in class. About a quarter of the students took this offer and worked on their lab courses, some with C# (version 4), some with C++. Not surprisingly, their solutions have been rarely as succinct and elegant as the X10 programs.

Tools are an important aspect and success factor for X10: Students found it convenient to work with the eclipse editor, which has been tailored to X10. This is especially so due to X10's advanced type system: Initially it took all participants a while to get used to typing with constraints. Some students felt that the eclipse X10DT platform was somewhat unreliable at the time of the course but also acknowledged that the robustness and speed of the plugin has improved significantly since then.

Finally, one student commented that "*... the language X10 should not be used in future classes, since parallel programming is simplified significantly, and for that reason one does not run into issues and problems that occur, when conventional programming languages are used for parallel programming*". This statement confirms that X10 permits to express problems with simple syntax - which was the motivation for using X10 in the first place. What nicer compliment could X10 developers hope on their language design?

5.3 Limitations

A critical comment at the end: As in most cases parallel programming is done for performance reasons, the course is missing a section on performance tuning. Currently, the course has a clear bias toward correctness issues of parallel programs; thus students are left largely 'performance illiterate'. All exercises were done using the Java backend of the X10 toolchain. In future iterations of the class, we plan to address this concern.

Since our report is about the first iteration of the class, we do not have insights on the medium and long term impact, e.g., whether students retain knowledge and reuse it in further classes or later during their professional life. By coincidence, one student reported to me several monthes after the first iteration of the parallel programming overview course, that the course greatly helped him during a subsequent internship, where he worked on a parallel simulation of heat dissipation in a data center.

6. Conclusions

In the discussion about introducing concurrency in the undergraduate curriculum, we propose a new design point: An overview class for concurrent programming, followed by specialized classes with focus on specific topics and technologies. The purpose of the overview class is to make students conscious about the different abstraction layers of parallel programming, teach each layer by examples and discuss correctness and performance issues at each layer. We believe this approach strikes a good balance between a class with sole focus on theory and principles, and a class with specific focus on selected technologies at a specific tier, commonly tier 3. The course we propose was taught to about 20 students with very positive feedback, e.g., about the spread of topics in class. In the first iteration of the class, emphasis was mostly on correctness issues; we plan to include a section on performance tuning in future classes.

Acknowledgments

I'd like to express my sincere thanks to Thomas Gross, Michael Scott, and the anonymous reviewers: This article greatly benefited from discussions with them and their diligent comments. I am especially grateful to the students who took the class during the summer 2010 for their committed participation and their patience with this experimental course. This work was supported by a grant from IBM.

References

[1] G. Agrawal. Exposing every undergradate to parallelism: Baby steps at Ohio State. In *Workshop on Integrating Parallelism Throughout the Undergraduate Computing Curriculum (CPATH)*, Feb. 2011.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance

computers. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, pages 2–11, Nov. 1990.

[3] Apache Foundation. Struts: a framework for building web applications. http://struts.apache.org, 2011.

[4] K. Asanovic, R. Bodik, J. Demmel, J. Kubiatowicz, K. Keutzer, E. Lee, G. Necula, D. Patterson, K. Sen, J. Shalf, J. Wawrzynek, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California Berkeley, 2007.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, M. Garzaran, B. B. Fraguela, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–57, Mar. 2006.

[6] R. L. Bocchino, V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, , and M. Vakilian. A type and effect system for deterministic parallel Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2009.

[7] R. Brown, P. Garrity, T. Yates, and E. Shoop. WebMapReduce: An accessible and adaptable tool for teaching map-reduce computing. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, Mar. 2011.

[8] B. Chamberlain, S.Choi, E. Lewis, C. Lin, L. Synder, and W. Weathersby. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, 1998.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commununications of the ACM*, 51:107–113, Jan. 2008.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, Mar. 2009.

[11] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, pages 89–99, Jan. 1989.

[12] D. Ernst, B. Wittman, B. Harvey, T. Murphy, and M. Wrinn. Preparing students for ubiquitous parallelism. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, pages 136–137, Mar. 2009.

[13] A. Grama, G. Karypis, , V. Kumar, and A. Gupta. *Introduction to Parallel Computing (2nd Edition)*. Pearson, 2003.

[14] T. R. Gross. Undergraduate class in compiler construction. ETH Zurich, 1999.

[15] T. R. Gross. Using a class on compiler design to teach software construction. Available at http://www.lst.inf.ethz.ch/research/publications/COMPILER_2000/COMPILER_2000.pdf, Sept. 2000.

[16] T. R. Gross. Breadth in depth. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, Mar. 2011.

[17] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Computational Science, 2011.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[19] K. Knobe. Ease of use with concurrent collections (CnC). In *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*. USENIX, Mar. 2009.

[20] L. Lamport. Specifying concurrent program modules. *Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[21] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2005.

[22] C. H. Nevison. Parallel computing in the undergraduate curriculum. *Computer*, 28:51–56, Dec. 1995.

[23] S. Rivoire. A breadth-first course in multicore and manycore programming. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, pages 214–218, Mar. 2010.

[24] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, , and D. Grove. Report on the experimental language X10 (version 2.1). http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf, Oct. 2010.

[25] M. L. Scott. Don't start with Dekker's algorithm: Top-down introduction of concurrency. In *Multicore Programming Education Workshop*, Mar. 2009.

[26] E. Shoop and R. Brown. Modules in community: Injecting more parallelism into computer science curricula. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, Mar. 2011.

[27] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL parallel container framework. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 235–246, Feb. 2011.