Workshop
on
Integrating Parallelism Throughout the Undergraduate Computing Curriculum

Abstracts and Schedule

Saturday, February 12, 2011
San Antonio, Texas

Held in Conjunction with the
2011 ACM Symposium on the Principles and Practice of Parallel Computing

copies of this can be found at

Sponsored by the National Science Foundation
CPATH program
All rights to the works contained in this document remain with the authors, and they should be contacted before duplication or other uses of these works. The compilation of these works in this document is copyright 2011 by Samuel P. Midkiff.
Workshop Schedule

8:30: Session 1

Opening Comments - Sam Midkiff

Keynote - Dick Brown (St. Olaf) *Teaching Parallelism to Every Undergraduate CS Student.*

David Padua (UIUC), *Parallel programming education at Illinois. The first twenty years*

10:00: Break

10:30: Session 2 (Gagan Agrawal, chair)

Ed Gehringer (NCSU), *Parallel computer architecture: Essentials for both computer scientists and engineers*

Michael L. Scott (University of Rochester), *Commonplace concurrency*

Uzi Vishkin (University of Maryland College Park), *Algorithms-based extension of serial computing education to parallelism*

12:00: Lunch

1:00: Session 3 (Ed Gehringer, chair)

Gagan Agrawal (Ohio State), *Exposing every undergraduate to parallelism: Baby steps at Ohio State*

David G. Wonnacott (Haverford College), "Be Ready For Anything" --- Preparing undergraduates for parallelism

Daniel J. Ernst (University of Wisconsin Eau Claire), *Tackling the logistical challenges of pervasive educational parallelism*

M. Manjunathaiah (University of Reading, UK), *CSP as a general concurrency model*

3:00: Break

3:30: Discussion (Sam Midkiff)

5:00: End of workshop
Integrating Parallelism Throughout the Undergraduate Computing Curriculum

Sam Midkiff, Purdue University (smidkiff@purdue.edu)

Parallelism, concurrency, and the theoretical and engineering issues surrounding these have long been interesting and intellectually challenging issues. With the widespread adoption of multicore processors and widespread availability of clusters, the topics of parallelism and concurrency have assumed a role of great practical and economic importance. This in turn has led to a re-evaluation of curriculums in computer engineering and science curriculums to ensure that these now crucial topics are receiving, if not the attention they deserve, as much attention as they can have given the constraints imposed by national standards and accrediting organizations, and the pressures of other demands on the curriculum.

The talks in this workshop will cover the efforts of educators in programs with a long-standing focus on parallelism, programs with novel techniques, programs with a primary focus on research and programs with a primary focus on undergraduate education. It is my hope that the interplay of these different efforts, and the discussions and contacts that come out of this workshop, will lead to a better understanding of these important issues by all of our students.

Finally, I would like to thank the National Science Foundation and Dr. Harriet Taylor for their long-standing support of these efforts. I would also like to thank the workshop presenters and participants -- the success of this workshop will be entirely a result of your efforts and interaction during the workshop itself, and in what you do after the workshop.
Teaching Parallelism to Every Undergraduate CS Student
Richard Brown
St. Olaf College (rab@stolaf.edu)

At long last, parallelism is about to blossom in undergraduate Computer Science (CS) education! Of course, such topics as concurrent programming and the use of parallelism in computer architecture have appeared in national curricular recommendations for CS majors since ACM’s Curriculum ’68, reaching something of a high point in the 1990s. For example, the ACM/IEEE joint curricular recommendations in 1991 called for at least three hours of instruction in parallel algorithms, and at least three more hours on distributed and parallel programming constructs, including study of the “promise of functional, logic, object-oriented or other special-purpose languages on highly parallel or distributed architectures,” accompanied by parallel programming experience in a language such as Ada, Concurrent Pascal, Occam, or Parlog. But few institutions incorporated this push towards parallel computing into their CS academic programs. (Note: I will use parallel computing as a blanket term, encompassing concurrency, parallel architectures, distributed systems, etc.) The 2001 ACM/IEEE joint curricular recommendations reduced the minimum requirement in parallel algorithms to zero hours, and dropped the knowledge unit on parallelism in programming languages altogether (although they did call for web-inspired client-server applications).

The industry shift to multi-core architectures for commodity computers means that software products cannot stay solely sequential for long and remain competitive in performance. Our customary exponential improvement in hardware performance will now primarily be achieved by multiplying the number of cores per computer, and this creates a natural mandate for virtually all CS students to learn more about parallelism. The lack of such a mandate undermined efforts to expand parallelism in CS curricula 20 years ago, in spite of the presence of an intellectual body of work that could have supported such an expansion. Fortunately, we have that foundation to build on now as we prepare for the challenge of teaching every undergraduate CS student more about parallel computation.

Such broad curricular change seldom happens quickly. But commodity multi-core computing has already been standard for years, so today’s undergraduates in CS already need an understanding of parallelism in order to be prepared for their careers. How can the CS education community make such a transition without delay? I will consider this issue at both the (inter)national movement level and the level of individual and collaborative contributions.

For ITiCSE 2010, a working-group of American and German researchers and educators considered strategies for responding to the need to teach more parallelism in CS curricula everywhere [report to appear]. We call for a spiral approach to teaching parallel computing that integrates
parallelism at all undergraduate levels, presenting principles of parallelism reinforced by hands-on computational experience. These pedagogical practices have proven effectiveness, and an “early and often” approach will help raise the profile of parallel computing in CS curricula. We also suggest a body of knowledge in parallel computing, organized into a framework that is motivated by applications in context and structured around a general problem-solving strategy for parallelism (software design choices; algorithms and data structures; software environment; hardware features), with conceptual issues and theoretical foundations associated throughout. Our findings are hardly “the last word.” Instead, we offer this study in hopes of sparking further discussion and expedited action within the international CS education community.

At a more applied level, Elizabeth Shoop of Macalester College and I are producing materials that take a modular approach to injecting parallel computing throughout undergraduate CS curricula, beginning with the introductory course [www.csinparallel.org]. We seek to create well-supported, flexible teaching modules that instructors can readily insert into various courses, accommodating a wide range of local pedagogical practices and preferences. For example, a Java-based laboratory exercise to construct a multithreaded web crawler might naturally introduce programming with threads and related implications of parallelism to students in various intermediate courses, depending on institution. In another module, parallelizing a loop using OpenMP on Intel’s Manycore Testing Lab facility [software.intel.com/en-us/articles/intel-many-core-testing-lab/] introduces notions such as race conditions and reduce operations to students with a minimal reading understanding of C/C++ programming. That MTL module can be used in elementary courses in computer organization or software design, or in higher level courses for introducing OpenMP programming and related issues in parallelism.

We introduce beginning students to parallelism computing primarily through a module using WebMapReduce (WMR) [webmapreduce.sourceforge.net/], a simplified interface to the open-source Hadoop implementation of fault-tolerant map-reduce computing on clusters. Map-reduce computing connects the student’s work to highly visible applications of data-intensive scalable computing, including Google services and computations related to popular social networks. WMR enables introductory students to program in the language they are studying for their course, and provides the power of scalable Hadoop computation with a greatly simplified interface. WMR has also proven useful in various advanced CS courses.

There remains the challenge of supporting and encouraging non-specialist instructors who incorporate parallelism in courses that have not traditionally included such topics. But a strategic, incremental approach that presents principles of parallelism together with hands-on experience, at all levels of the CS curriculum, offers the best hope for teaching CS undergraduates the parallelism they now need.
Parallel programming education at Illinois. The first twenty years
David Padua
University of Illinois at Urbana-Champaign (padua@illinois.edu)

I will discuss my experience in teaching a parallel programming class for scientist and engineers that is part of our "Computational Sciences" curriculum. This course has been taught regularly since the late 1980s. I will also discuss two courses we have recently developed: One on program optimization techniques and another on parallel programming for CS majors. Finally I will describe other courses in parallel programming developed by my colleagues at Illinois on parallel numerical algorithms and theory of parallelism.
Parallel Computer Architecture: Essentials for both Computer Scientists and Engineers
Ed Gehringer and Yan Solihin
North Carolina State University {efg, solihin}@ncsu.edu

Introduction

Parallel computer architecture is a topic not covered well by mass-market textbooks, but one that is nonetheless essential to a well rounded education for both computer engineers and computer scientists. For computer engineers, it represents the macroarchitecture side of system design, and complements a course in microarchitecture, or processor architecture. For computer scientists, parallel architecture is by far the most relevant level of architecture, because it affects the programming model.

NC State’s CSC/ECE 506, Architecture of Parallel Computers, fills this need. As an introductory masters-level course that is also available to senior undergraduates, it enrolls nearly 100 students per year. It is a core course for masters programs in both computer engineering and computer science. The textbook for the course is the locally written Fundamentals of Parallel Computer Architecture: Multichip and Multicore Systems [1] by Yan Solihin. The text takes an approach that explains basic concepts in an intuitive and holistic way, in order to help students understand how these concepts relate to one another. It leaves out details that may hinder such understanding. The main topics of the course are (i) models of parallel programming, (ii) parallel programming techniques, (iii) architectural support for synchronization, (iii) cache coherence, both bus-based and distributed shared memory, and (iv) memory consistency. Though the course is not a parallel programming course, an important component is writing shared memory programs with OpenMP, with matrix-based applications as well as applications that use linked data structures.

The students

At least three-quarters of the students in CSC/ECE 506 are graduate students, with the remainder being senior undergraduates. NCSU has large masters programs in both computer science and computer engineering. The programs are of similar size, enrolling a total of more than 200 students per year. A substantial number of electrical engineering students also take the course; however, they are usually no more than 10% of the class. Computer engineering students take two entry-level architecture courses, CSC/ECE 506, and ECE 521, which is based on Hennessy and Patterson’s Computer Architecture: A Quantitative Approach. The courses are designed so that either course may be taken first, or they may be taken at the same time.

The course is also a core “systems” course for computer science masters students. They have a choice among seven courses to fulfill this requirement. Thus, computer science students are usually less numerous than computer engineering students, though their numbers have been trending up with the increasing attention given to parallelism. While CS students may take the course
anywhere during their graduate program, computer engineering (CpE) students usually take it in the first year because it is a prerequisite for two advanced ECE courses—ECE 706, Advanced Parallel Computer Architecture, and ECE 743, High-Performance Computer Architecture.

## Course organization

Table 1 shows the organization of the course. It begins with a consideration of the three main parallel-programming models, shared memory, message-passing, and data-parallel. Following that, we discuss how programs are decomposed for parallel execution, using constructs such as DOALL, DOACROSS, DOPipe, and function parallelism. At this point, OpenMP is introduced. We explain why loop-level parallelization is insufficient for linked data structures, and describe parallelization among readers and locking approaches.

We proceed to a short overview of cache organization, a topic that is covered in more detail in our processor-architecture course, ECE 521. From here, we move directly into cache coherence, first to invalidation-based protocols (such as MSI and MESI) and then update-based protocols (e.g., Dragon). Hardware support for locking is our next topic; we cover lock and barrier implementations. From here we go on to distributed-memory machines, paying particular attention to table-based coherence protocols. Interconnection networks are the last major topic of the course.

<table>
<thead>
<tr>
<th>Table 1. Course organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models of parallel computation</td>
</tr>
<tr>
<td>DOALL, DOACROSS, etc. for parallelization</td>
</tr>
<tr>
<td>Data-parallel programming</td>
</tr>
<tr>
<td>Parallelizing linked data structures</td>
</tr>
<tr>
<td>Cache coherence (bus-based)</td>
</tr>
<tr>
<td>Locking</td>
</tr>
<tr>
<td>Memory consistency</td>
</tr>
<tr>
<td>Caching and distributed memory</td>
</tr>
<tr>
<td>Interconnection networks</td>
</tr>
<tr>
<td>Note: 75-minute classes, two per week</td>
</tr>
</tbody>
</table>

## Ancillary materials

The textbook has a Web site at [http://www.cesr.ncsu.edu/solihin](http://www.cesr.ncsu.edu/solihin). It contains presentation slides for all of the first twelve chapters (which are the chapters covered in this course). One important component of the slides are PowerPoint animations of several cache-coherence algorithms, illustrating how various patterns of sharing are reflected in cache state. Figure 1 shows a snapshot from the MSI protocol.

In addition to materials currently on the Web site, Dr. Gehringer’s section is developing a wiki supplement to the textbook [2], covering additional topics and describing how ideas described in the text are realized in current parallel machines.

## Conclusion

We believe that this course fills an important need in the curriculum by stressing how parallel programming and parallel architecture affect one another. Writing efficient parallel programs has always required an understanding of the architecture on which the programs run, and designing an efficient parallel architecture requires an understanding of how parallel programs are written.
Such courses have long suffered from a paucity of textbooks that cover the nuts and bolts of parallel computers, especially ones that present concepts at a uniform level to beginning graduate students. Students who take CSC/ECE 506 are well positioned to go on to advanced courses in parallel architecture (e.g. design of current multicore systems), parallel programming, and computational science.

![Diagram of Processor $P_1$ Reading A]

**Figure 1.** Cache snooping in the MSI protocol

We would be pleased to share our materials with others who wish to extend their curricula in this direction. This includes the textbook, which is self-published in order to be available at modest cost, the slides, including animations of many coherence protocols, several programming assignments, and the wiki supplement, which extends the breadth of the text and keeps it up to date.

**References**


Biographies

Ed Gehringer is an Associate Professor of Computer Science and Computer Engineering at North Carolina State University. He has been the chief organizer of the Workshop on Computer Architecture Education for the past 10 years, and also chaired the SPLASH Educators' Symposium last October, which focused on parallelism in the curriculum.

Yan Solihin is an Associate Professor of Electrical and Computer Engineering at North Carolina State University. He is a well known researcher in computer architecture who was General Chair of HPCA 15. He is the author of Fundamentals of Computer Architecture: Multichip and Multicore Systems, a parallel-architecture textbook used in several programs around the US.
Historically, most undergraduates have been introduced to concurrency and parallelism in a course on operating systems. Typically this introduction has been motivated by the need for concurrent interrupt handling and, more recently, for parallel kernel execution on multicore processors. As a result, coverage has tended to focus on low-level synchronization mechanisms suitable not only for concurrent data structure updates, but for implementation of the very threads that are performing the updates.

From our current perspective, six years into the multicore revolution, this historical pattern has had at least three unfortunate consequences. First, it has put concurrency and parallelism late in the typical student’s career. Second, it has conveyed the impression that anything to do with concurrency and parallelism is almost impossibly difficult. Third, it has favored the development of notation, tools, and syllabi appropriate for kernel hackers, but not for more typical programmers.

I have argued that concurrency and parallelism should be thought of not as an esoteric specialty, covered in a special course, but as a routine aspect of programming, incorporated into every task and topic for which it is appropriate. So yes, low-level synchronization needs to show up in the OS course, and memory models and coherence protocols need to show up in computer architecture. At the same time, concurrency should appear as one of the standard control flow paradigms in the programming languages course, alongside iteration and recursion; event-driven programming should appear in HCI and web programming courses; data-parallel programming should appear in graphics and scientific programming (at least); and simple collateral execution (to use the old Algol 68 term) should appear in any algorithm in which asymptotically significant computations are independent of one another (think quicksort). And of course libraries and tools that run in parallel internally should be used even in the Computer Appreciation course.

Covered in context, with appropriate tools, and at an appropriate level of abstraction, concurrency and parallelism need not be intimidating. A year ago, I taught a CS2 (Data Structures) course for non-majors, using Python as the language of instruction. I included a lecture unit and project on event-driven programming. Drawing inspiration from Kim Bruce’s experience at Pomona College, I made the project an implementation of the popular “Frogger” game. I didn’t tell students that synchronization was difficult, and they didn’t find it so: they had much more difficulty, for example, with earlier units on backtracking and dynamic programming.

To support the use of concurrency and parallelism throughout the undergraduate curriculum, we will need appropriate notation and tools, including language-level support for the easy cases (parallel independent for-all, safe futures, collateral execution); data race finders; model checkers; and production-quality build, test, and debugging environments. In the past we had the luxury of teaching to the experts; now we need to teach to everyone.
Algorithms-based extension of serial computing education to parallelism

Uzi Vishkin, University of Maryland Institute for Advanced Computer Studies (vishkin@umd.edu)

Parallel computing provides now the only avenue for continued improvement in performance in general-purpose computing. The basic transition to parallelism appears to be robust: the mainstream hardware and software computing industries have been forced to bet their future on parallelism. But, how the commodity parallel general-purpose computer of the future will be built and programmed for performance remains unclear. Points of reference include:

- The programmer of today’s parallel machines must overcome several ‘productivity busters’ beyond just identifying operations that can be executed in parallel:
  - impose the [CS99] 4-step programming-for-locality recipe: decomposition, assignment, orchestration, and mapping, which is often difficult;
  - reason about concurrency, including race conditions, in threads;
  - for machines such as GPU, that fall behind on serial (or low-parallelism) code, whole programs must be highly parallel.

- The National Research Council report [FM10] points out that while heroic programmers can exploit today vast amounts of parallelism, whole new computing “stacks” are required to allow expert and typical programmers to do that easily.

- None of 40+ students in a fall 2010 joint UIUC/UMD course got any speedups using OpenMP programming on simple irregular problems using an 8-processor SMP, but they got 8X-25X speedups on XMT.

- SMPs do not scale beyond 8-16 processors and it is not clear how well other cache coherence solutions work due to their high overheads.

- Ease-of-teaching comparison by DARPA-HPCS-funded software engineering experts Basili and Hochstein (UMD) [HBVG08] showed XMTC/PRAM development time is half of MPI.

Beyond introductory programming, standard CS curriculum emphasizes data-structure and algorithms courses over programming ones. This fact and the noted lack of clarity about the future led me to devote class time to algorithms over programming and favor teaching the simplest common denominator of current approaches.

The education platform I developed is based on the following elements:

1. Identify ‘thinking in parallel’ with the basic abstraction behind the [SV82b] work-depth framework. This framework was previously adopted as the presentation framework in 2 PRAM algorithms texts: [J92, KKT01].

2. Teach as much PRAM algorithms as timing constraints and developmental stage of the students permit; extensive ‘dry’ theory homework is required from graduate students, but little from high-school students.
3. Students self-study programming in XMTC (standard C plus 2 commands, spawn and prefix-sum) and do demanding programming assignments.

4. Provide a programmer’s workflow that links the simple PRAM abstraction with XMTC programming. The synchronous PRAM provides ease of algorithm design and reasoning about correctness and complexity. Multi-threaded programming relaxes this synchrony for implementation. Since reasoning directly about soundness and performance of multi-threaded code is known to be error prone, the workflow assigns a much simpler task to the programmer: establish that the multi-threaded program behavior matches the PRAM-like algorithm it implements.

5. Unlike the PRAM theory, XMTC is far from ignoring locality. Unlike today’s common approaches, XMTC preempts the harmful effect locality has on programmer’s productivity.

6. If the XMT architecture is presented, it is done only at the end of the course; students don’t learn serial architecture prior to learning serial programming, so why should they learn parallel architecture?! (However, parallel architecture relevant to OpenMP and MPI had to be taught in the UIUC/UMD course.)

Experience:

**K-12** Since 2007, various snippets of the approach were taught mostly by two high school teachers to more than 100 middle school and high school students of a wide range of backgrounds. Among them were students from Montgomery Blair, Maryland and Thomas Jefferson, VA, magnet high schools, Baltimore Polytechnic high school, whose student body is 70% African-American, and a Montgomery County, MD Public Schools middle-school summer workshop for children from underrepresented groups. Teacher S. Torbert (TJHS) self-taught himself from publicly available material. Teacher D. Ellison (Math Ed, PhD student, U. Indiana) who taught at the Baltimore high school and the middle-school workshop was advised by Math Ed professor R. Tzur (Purdue/U. Colorado), an expert in learning as understood in education.

**College freshmen** A class that included 3 sorting programming assignments and one for finding the median – a proper load for a freshmen serial programming course – was taken by 19 students, mostly non-CS majors. This overall K13 experience was reviewed in: (i) [VTETC09], a keynote at CS4HS’09@CMU and (ii) [TVTE10], a SIGCSE’10 paper that reports a decisive teaching advantage of XMT over CUDA, MPI and OpenMP at TJHS.

**Graduate class** included standard PRAM algorithms plus 6 XMTC programming assignments including the [SV82a] graph connectivity (done also by a couple of Blair 10th graders) and performance tuning.

**Senior level course** taught less theory. An inter-university senior-level course using teleconferencing with UIUC included 7 lectures on PRAM/XMTC, demonstrating a useful role for them in a course that included 3 joint OpenMP /XMTC programming assignments, as well as MPI.

**Course on general algorithms** Several PRAM classes were included.

available for free download along with extensive documentation. However, many students programmed a 64-processor XMT FPGA machine. 2. Extensive teaching material including class notes, over 31 hours of video-recorded classes, and a day-long tutorial are also available on-line.

Though quite different, like-minded approaches include Cilk [CLRS09] and NESL [B96].

References


[HBVG08] L. Hochstein, V. Basili, U. Vishkin and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. Journal of Systems and Software, 2008. This paper compares the programming effort in MPI and XMTC for a similar programming assignment. The main finding is that with high confidence XMTC development time is nearly half of MP.


[V11] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. CACM 54,1 (January 2011), 75-85. This paper provides a good introduction to the PRAM abstraction and the XMTC programmer’s workflow.

Exposing Every Undergraduate to Parallelism: Baby Steps at Ohio State
Gagan Agrawal, Feng Qin, and P. Sadayappan (agrawal@cse.ohio-state.edu)

As Ohio State University is converting from quarters to semesters starting from the academic year 2012-13, the computer science department went through the process of redesigning the undergraduate curriculum during the school year 2009-2010. As part of the final outcome, we are pleased that we have been able to integrate parallelism and locality as part of the undergraduate core: every undergraduate will be doing one lab in parallelizing an application (using Pthreads, and on a small multicore machine). Closely related to parallelism, each undergraduate will also be doing an exercise in managing locality of programs.

The department level process we went through highlighted many challenges in developing computer science curricula, i.e contention for topics to be included in the undergraduate core, the pressures of keeping the undergraduate core small, and conflicting views among the faculty.

In the end, the systems faculty in the department had to decide what are the most important topics to be covered with 6-7 credits of required systems coursework (choosing among topics like C programming, assembly programming, machine representation of data and instructions, operating system concepts, computer organization and architecture, and any other topics we would like to include). Needless to say, this imposed a difficulty in adding any material to the systems core that has traditionally not been part of the CS core. In the end, the following systems courses were agreed upon: 1) Systems I: a class that students will take in their fourth semester, which will cover C programming (our 2 semester Intro sequence will be in Java), computer organization, data and instruction encoding, and assembly programming, and 2) A junior level course, Systems II, which is designed by extending our 10-week required operating systems course in the quarter system to include parallel programming and management of data locality at the application level.

While parallel programming and application-level locality management are not traditional OS topics, and clearly there were reservations about introducing these topics (as opposed to spending more time on topics like device management and kernel design), we felt that this was our best opportunity to have every computer science undergraduate get exposure to parallelism and locality. By introducing parallelism as an extension to process synchronization, we can also leverage on Pthreads that students work with for producer-consumer kinds of problems.
“Be Ready For Anything” - Preparing Undergraduates for Parallelism

David Wonnacott, Haverford (davew@cs.haverford.edu)

The sheer variety of tools and paradigms for parallel programming poses a challenge for educators: we must introduce all students to the increasingly important concepts of parallelism, without yet knowing which tools and paradigms will be used by the student (or even the industry) in the near future. We see this as yet another reason to emphasize intellectual flexibility as one of the hallmarks of true education. The cornucopia of approaches to parallelism is simply a rich set of opportunities to expand a student's palette of paradigms.

This view is best supported by an environment in which strong connections can be created among courses. The process of relating new paradigms in upper-level courses to those explored in introductory courses reinforces foundational concepts while developing new material. This is particularly true if the anticipation of such connections influences the way we develop our foundation.

I will illustrate these principles with selected examples from Haverford's curriculum, primarily focusing on those that appear in our operating systems course, as I'm teaching that this semester. I plan to discuss:

1. Our development of certain contrasts in the introductory semester, specifically (a) contrasting the pure functional and imperative paradigms and (b) contrasting static and dynamic views of programs, to facilitate the jump to thinking about threads and data races in our 300 level operating systems course;

2. Our definition of "computational complexity" in general terms that help our students grow in sophistication of this subtle topic, from "Complexity is about measuring computation cost in many ways: for now let's count how many times these instructions are executed." in introductory year, to "Let's distinguish total number of functional units from the length of the longest path through those units." in our 200-level courses on hardware and computer arithmetic, to "How do we balance communication/synchronization costs and concurrency benefits?" in 300-level courses on operating systems and on distributed memory parallelism;

3. Our emphasis on expressing and maintaining invariants, from foundations in software engineering in our introductory courses through techniques for managing complexity in shared-memory parallelism at the upper level.

These examples are skewed toward the threads-and-locks paradigm that comes to the fore in operating systems design, but other parallel paradigms are developed across our curriculum. For example, parallel traversal of data structures is discussed briefly in the second semester data structure course and reappears in courses on programming language design/compilers (including the map-reduce paradigm and other forms of tree rewriting) and analysis of algorithms (including parallel "pointer jumping" and parallel prefix operations or generalizations thereof).
Tackling the Logistical Challenges of Pervasive Educational Parallelism
Dan Ernst, University of Wisconsin - Eau Claire (ernstdj@uwec.edu)

In the last few years, curricular interest in parallel computing has been growing quickly, as evidenced by a steep increase in papers, articles, and workshops. A few years ago, I initiated changes in the curriculum at the University of Wisconsin - Eau Claire to attempt to address the parallel deficiency in our curriculum. The process of implementing these changes, along with my engagement with other institutions considering similar things, has highlighted a number of challenges that impede the progress of integrating parallelism into undergraduate programs. These challenges include issues as diverse as the significant amount of course adaptation required, the relative unreadiness of faculty, the scarcity of educational resources and tools, and the relative uncertainty of future programming models and methods.

To address these challenges, action will likely need to happen at levels beyond independent faculty decisions - many require department-level or community-wide initiative. In my talk, I will discuss the causes and consequences of these challenges using small case studies, and present some possible solutions to the logistical hurdles faced by faculty.

Biography:
Dr. Daniel Ernst is an Assistant Professor of Computer Science and Director of the Center for Computational Science at the University of Wisconsin - Eau Claire, where he studies computer architecture, parallel computing, and computational science, with a focus on curricular and pedagogical issues related to parallelism in undergraduate computer science programs. His previous research has examined high-performance, low-power, and fault-tolerant microarchitectures.

Dr. Ernst is also a core founding member of the Educational Alliance for a Parallel Future (EAPF), a industry/academic partnership dedicated to establishing parallelism as a core competency within computer and computational science education and practice. As part of this group, Dr. Ernst is also an author/editor of the new ACM Parallel Computing Tech Pack, a curated meta-resource for computer scientists interested in learning parallel programming.
CSP as a general Concurrency model
M. Manjunathaiah, University of Reading, UK (m.manjunathaiah@reading.ac.uk)

To develop an appreciation of concurrent system design and implementation, we explore a model of concurrency based on Communicating Sequential Processes in a final year optional module called ‘Concurrent Systems’ (http://www.reading.ac.uk/module/0910/SC/ CS3H7.htm). In the spirit of being ‘architecture neutral’ we develop general concepts of concurrency using CSP in which the principle method of co-ordination of concurrent components is through communicating events and the underlying idea that any system design can then be completely characterized in terms of its communicating behavior. The flavor of the course is mainly on developing rigorous notions of semantic models to capture system properties for both design and verification - similar in style to a course at Oxford University Computing Laboratory. In particular we explore Traces and Failure-divergence models which gives students the depth of understanding to characterize deterministic and non-deterministic systems.

In order to bridge the theory with practice we have two main course works: 1) to develop modeling and implementation skills using CSP. Using a ‘real-world’ example of an Airport Car Park booking system, students develop a design and implementation of a simplified model of the system using their preferred choice of implementation languages JCSP, PyCSP or Jibu, and 2) to develop analysis skills students explore the use of refinement tools such as FDR. In addition students undertake a group study to explore an application area of CSP drawn from research literature to extend their knowledge on the scope and power of the model.

In this talk I will explore the above aspects of the course and contrast it with other courses which have specific focus such as CSP for multi-core programming.

Biography
Dr. M. Manjunathaiah is a Lecturer (equivalent to Asst. Professor) in Computer Science and Informatics at the University of Reading in U.K. His research work and interests are in massively parallel computing models, scalable parallel computing architectures, tools and techniques for practical parallel programming and formal methods for hardware software designs. He received a Master of Engineering degree from Indian Institute of Science, Bangalore, India. He was awarded the Commonwealth Scholarship in the U.K., to pursue a Doctoral programme in computer science. He received a PhD degree from the University of Southampton, U.K., in 1997 for his work on Compilation Techniques for Parallelization and Parallel Program Verification. After his doctoral studies he held research fellowships at the University of Reading, Oxford University Computing Laboratory and University of Manchester all in the U.K., researching into Polyhedral model, hardware-software co-design in the CSP model and dynamic parallelism.