Strategies for Preparing Computer Science Students for the Multicore World

Richard Brown
St. Olaf College (USA)
rab@stolaf.edu

Elizabeth Shoop
Macalester College (USA)
shoop@macalester.edu

Joel Adams
Calvin College (USA)
adams@calvin.edu

Curtis Clifton
Rose-Hulman Institute of Technology (USA)
clifton@rose-hulman.edu

Mark Gardner
Virginia Tech University (USA)
mkg@vt.edu

Michael Haupt
Hasso-Plattner-Institut, University of Potsdam (Germany)
michael.haupt@hpi.uni-potsdam.de

Peter Hinsbeeck
Intel Corporation (Germany)
peter.hinsbeeck@intel.com

ABSTRACT

Multicore computers have become standard, and the number of cores per computer is rising rapidly. How does the new demand for understanding of parallel computing impact computer science education? In this paper, we examine several aspects of this question: (i) What parallelism body of knowledge do today's students need to learn? (ii) How might these concepts and practices be incorporated into the computer science curriculum? (iii) What resources will support computer science educators, including non-specialists, to teach parallel computing? (iv) What systemic obstacles impede this change, and how might they be overcome? We address these concerns as an initial framework for responding to the urgent challenge of injecting parallelism into computer science curricula.

Categories and Subject Descriptors

K.0 Computing Milieux/General; K.3.2 Computer Science Education

General Terms

Algorithm, Design, Performance, Reliability, Theory

Keywords

Parallelism, Multi-core Computing

1. INTRODUCTION

1.1 The Multicore Revolution

We are motivated to teach more parallelism and concurrency in CS courses because of the necessary shift by hardware manufacturers towards multi-core computer design. For over four decades, hardware performance has improved at an exponential rate, a remarkable feat that derives from ever increasing miniaturization of transistor components. Since transistors comprise the *cores* in a computer, i.e., the circuits within a computer's central processing unit (CPU) that are capable of executing computer instructions, tinier transistors made it possible to create more and more powerful cores, in terms of both logical capability and speed. This continued until the last decade, when the industry began to encounter physical limitations on the speed of a single core. Since that time, computer performance has improved primarily by increasing the number of cores per computer rather than the speed of a core. This trend will continue for the foreseeable future. Nowadays, nearly all new computers provide at least two cores, and computers with 32 or more cores can be constructed with commodity parts. In a few years, "manycore" computers with hundreds or thousands of cores per computer will become available [9].

Before the emergence of multi-core commodity computing (approximately 2006), sequential software typically enjoyed performance improvements as a direct consequence of hardware performance improvements. Improvements came in both speed and *parallelism*, in which multiple computational actions occur at the same time. This parallelism within a CPU was largely out of the view of a programmer, although the software that programmers produced automatically benefited from it. But now, programmers will have to write software that explicitly and correctly takes advantage of parallelism from multiple cores operating simultaneously, in order to obtain the hardware speedups available on multi-core computers. Software products and systems that do not capitalize on the additional cores in a computer will be at a competitive disadvantage against those that make effective use of multi-core parallelism.

Programmers in the scientific and high-performance computing sectors have taken advantage of parallelism for decades, using specialized "supercomputer" hardware and/or networked clusters of computers. More recently, Internet-related companies have become enormously successful by creating convenient ways to program in parallel using clusters, as in Google's use of the map-reduce programming model [39]. But with the arrival of multi-core computing, general consumer software products will now be programmed for parallelism. As we have seen with previous improvements in commodity hardware performance, we can expect enhanced features requiring multi-core computing capabilities in coming versions of operating systems, productivity software, web browsers, and other products. Other applications, instead of capitalizing on increased performance, will take advantage of multi-core systems to offer the same level of service with lower power consumption. For example, mobile phones will soon have multi-core CPUs in order to operate at lower frequencies and prolong battery life [119][108]. Thus, in every

ITiCSE-WGR'10, June 26–30, 2010, Bilkent, Ankara, Turkey.
Copyright 2010 ACM 978-1-4503-0677-5/10/06...$10.00.

computing sector, programmers and system designers must now gain a deep understanding of parallel computing in order to obtain greater or even the same performance from a multi-core computer as single-core machine once offered.

This makes it urgent for computer science (CS) students to gain broad exposure to concepts of parallelism. The parallel features of a multi-core computing system represent a new kind of resource available for solving problems with computing. CS graduates who have studied parallelism will bring understanding of that technology to their careers, and employers will increasingly demand it. In some countries, curricular change can occur instantly through centralized decision-making; for example, as Intel's Michael Wrinn points out, all CS students in China now study parallelism [126]. Broad-based curricular change is more challenging in countries where curriculum decisions are made more locally.

1.2 Strategies For Teaching Parallelism

Therefore, we formed our working group in order to develop strategies for expediently and effectively incorporating parallelism into undergraduate CS courses and curricula. It is not our purpose to formulate a call for bringing more parallelism into CS education, since industry and academic leaders have already made that case compellingly [14][125][9]. Instead, our mission is to consider how the CS education community can respond strategically, productively, and as quickly as possible.

In order to explore our topic fully, our team includes not only CS educators from many types of academic institutions, but also from national research laboratories and from the hardware industry. Our disciplinary areas of expertise range from low-level hardware architecture to high-level virtual systems and programming languages. The results of our study will hopefully spark further discussion and expedited action within the international CS education community.

Since we seek to advance global change, we must choose an approach to our task that is neutral to differences in disciplinary viewpoints and institutional nature when possible. For example, we intend to highlight key concepts and issues with potential relevance at many points in a CS curriculum, such as the notion of scalability, believing that such pivotal concepts and issues are likely to apply in most curricular and institutional settings. Thus, our silence on a particular topic or tool should not be interpreted as disapproval. Likewise, we desire to make no assumptions about the curricula and institutions we address. For instance, we avoid relying on generic course names ("CS1", "CS2", "Algorithms", etc.) when describing where a parallel topic might appear in a CS curriculum, recognizing that there may be vast differences in the content and nature of courses that share the same title at different institutions.

We will use the word *parallel* as a generic term, meaning that multiple computer actions occur at the same time, whether physically or through software concurrency. Under this convention, we may think of concurrency as *virtual parallelism.* Writing "parallel" is more convenient than frequently writing, for example, "parallel or concurrent," but adopting a generic term is also strategic for our purpose. Having to master the technical distinctions and relationships between physical parallelism, concurrency, distributed computing, multi-core computing, grid computing, etc., constitutes a barrier to entry for non-specialists, whether they are students or faculty. If beginners may safely use "parallel" to mean any of the above, we can all move on toward the primary goal of introducing everyone to parallelism. Furthermore, in practice, implementation choices between virtual or physical parallelism, or multi-core vs. distributed computing vs. a hybrid system, change quickly based on the resources available in a particular system. Thus, "parallel" as a generic term is useful even for practitioners and specialists. We note that even if the computing industry were not shifting to multi-core computation at this time, we might now be conducting an ITiCSE working-group strategic study in response to the explosion of distributed computing (e.g., Google's use of map-reduce computation [39]).

Our document has a bias towards teaching principles of parallelism using experiential learning. We affirm the research literature endorsing active learning strategies, even though the present-day tools students may use to get their hands-on experience may soon pass away. We also believe that students who learn the most enduring parallelism principles available will be best equipped for the imminent flow of new tools and techniques that will emerge during this new wave of parallel computing.

1.3 Overview

The following sections of this document consider strategies for introducing parallelism in CS curricula by examining several aspects of such strategies.

In Section 2, we present a framework for expressing parallelism content that integrates both an educator's goals and a practitioner's objectives. In Section 3, we explore teaching and learning practices, including the issue of when and where to teach concepts of parallelism within a CS curriculum. In Section 4, we examine goals for online sharing of content materials, teaching strategies, assignments, and feedback among a supportive community of CS educators. In Section 5, we suggest two guidelines for bringing about the timely systemic change of injecting parallelism into CS curricula, and briefly addresses some of the challenges presented by such a transformation. We summarize our findings and suggest further strategic steps in Section 6.

2. TOWARD A PARALLELISM BODY OF KNOWLEDGE

Parallelism and related fields of study have a long and deep tradition. In addition to the underlying lasting concepts, there are also a variety of tools and techniques for realizing parallel computation. In this section we attempt to catalog many of the lasting concepts that students should be exposed to. We do not offer these as a comprehensive list, but as a starting point for identifying a central body of knowledge in parallelism that could be included in undergraduate curricula. Along with this catalog of ideas, we identify some of the existing tools that instructors might use to help their students explore these concepts.

In choosing the topics to include in a particular curriculum, the overriding concern should be to help our students learn to "think in parallel". By that, we mean that graduates should be able to:

- recognize the opportunities for parallelism in any problem and
- evaluate the applicability of different parallel solution strategies.

Different institutions will choose to include different concepts in their curricula. In section 3, we suggest some strategies for choosing from the concepts we describe in this section and integrating them into existing curricula.

We found it useful to organize our catalog of concepts into a framework that incorporates both potential curricular structure and practical development of applications, summarized in Table 1.

Table 1. Organizing the body of knowledge in parallelism.

Motivating Problems and Applications (2.1)	Software Design (2.2.2)	Conceptual Issues and Theoretical Foundations (2.2.1)
	Data Structures and Algorithms (2.2.3)	
	Software Environments (2.2.4)	
	Hardware (2.2.5)	

In this framework, we organize topics into four broad parallelism knowledge areas, represented by the central column in Table 1. Problems and application areas provide motivating context for students to study and use solutions from these knowledge areas. Cutting across these knowledge areas are the basic, well-known conceptual issues and theoretical foundations in parallelism, which we treat as a separate knowledge area.

This leads us to the following *goals for CS graduates in this new age of parallelism*.

Given a problem to solve, students should be able to choose and implement the following:

- The software design that will ensure efficient and reliable use of parallelism,
- The algorithms and data structures that will work best,
- The software environment to use, and
- The parallel hardware to use,

while keeping in mind the issues that will arise from those choices and the amount of scalability and performance that will be achieved.

2.1 Problems as Motivation and Context

Students are more motivated and develop a greater understanding when they see how a concept applies in context [16]. Here we provide some sample problem domains that serve as natural motivators for the use of parallelism.

Web search is a parallel application that students use frequently. Google, the well know search engine, utilizes massive amounts of essentially commodity PCs in large datacenters to create, replicate, and constantly update the index of the world-wide-web that millions of people search every day [19].

Another real world problem that shows the need for parallelism is sequence searching in bioinformatics. Scientists compare unknown gene or protein sequences against a database of known sequences to determine similarities, which help identify functionality. The database of all known genes, their nucleotide sequences and associated information, GenBank, is currently 437 GB in size [11]. Metagenomics projects, such as the Global Ocean Survey [122], and new technologies for generating DNA sequences are expected to dramatically accelerate the amount of data produced [99]. Indeed, such projects routinely use parallel techniques to produce gene sequence data by employing large number of sequencing machines running in parallel [124]. The time it takes to compare an unknown sequence to the database has become impractical for sequential computations whereas in 2003 a parallel version reduced the time from 22.4 hours to 8 minutes when run on 128 processors [38].

Image and video processing provides a rich array of example application areas that could motivate students. Single images themselves represent rich information that can be processed in parallel, and an increasingly more important application area is the processing of high-dimensional sets of images. Past work in this field drove a great deal of study of parallel architectures, and more recently this field is turning towards the use of stock parallel hardware [89]. For example, Colombo, et al. describe a system for tracking full human body motion to create realistic movie animation using a cluster of workstations, each responsible for particular tasks[28]. Image processing and analysis for robot vision systems is another motivating example for students [75]. In medicine, medical imaging is another example area where parallel hardware and algorithms are now being used to speed up the time to process scans and to enable patients to get results of cancer scans during an office visit [35]. Lastly, video processing for surveillance systems is a field that relies on parallel and distributed processing to provide real-time video analysis results to users [102].

Current financial analysis software takes advantage of parallel processing to provide analysis of market segments, such as equity options, in real time [34]. Many parallel algorithms for financial analysis have been proposed [63][81], which provide solutions in a real-world application domain of interest to many students.

An overview of many application areas is provided in [42], including these additional examples that instructors might use as motivation:

- Fluid dynamics
- Environment and Energy
- Molecular dynamics in computational chemistry
- Ocean Modeling
- Simulations of earthquakes
- Data mining
- Tree-structures: online massively parallel games
- Graphics: image processing, rendering, hidden-surface removal
- Sorting
- Ray tracing
- Weather/climate modeling
- Natural language processing and text analytics
- Speech recognition

2.2 Knowledge Areas

The following subsections discuss the cross-cutting theoretical concepts and each of the four knowledge areas shown in Table 1. We identify some representative learning outcomes for each. These are high-level goals that represent what undergraduate programs may strive to achieve and can be expanded to further detail for individual courses. We provide them as examples to stimulate thought, rather than as a prescriptive solution. Following the outcomes, we discuss some of the central ideas in each area—this is not intended to be complete, but to provide a starting point for further consideration when incorporating parallelism to a program. We conclude each knowledge area by suggesting a few other ideas that an institution might choose to cover in addition to (or instead of) the ideas we have chosen to discuss.

2.2.1 Conceptual Issues and Theoretical Foundations

Representative Educational Outcomes:

Computer science graduates will be able to:

- Identify and discuss issues of scalability in parallel computational settings.
- Define and recognize common types of parallelism and communication, namely data parallelism, task parallelism, pipelining, message passing, and shared memory communication.
- Define race condition and deadlock; identify race conditions in code examples; identify deadlock in computational and non-computational scenarios.
- Assess the potential impact of parallelism on performance using Amdahl's and Gustafson's Laws.

Central Ideas

One aspect of becoming educated in parallelism is having an awareness of key concepts and issues that arise in many contexts in parallel computing. Many of the technical details of hardware and software capabilities and environments change rapidly, but the concepts and issues in this category change much more slowly, and may be viewed as enduring principles that will continue to have relevance and value in the long term.

For example, the notion of *scalability* is central throughout parallel computing, and the behavior of an algorithm or computing strategy as some factor in a problem increases (for example, the number of processors or the size of the data) is often a key consideration when computing in parallel. Issues of scale can be identified and explored at all levels of a CS curriculum. Indeed, letting beginning students experience for themselves the effects of scale on computational time in various contexts can be an engaging and effective motivator for them to explore parallelism, while simultaneously imprinting invaluable lessons about the nature of parallel computation. Encounters with scalability in subsequent courses continue to reveal manifold aspects of that pervasive issue. In fact, an awareness of the notion of scale and a habit of considering issues related to scalability in computing scenarios are valuable components of any CS student's training, whether or not those scenarios happen to arise in the context of parallel computing. Scalability merits consideration whenever addressing a new computing situation, application, or system, along with other central issues such as algorithmic approach, efficiency, and correctness.

Besides knowledge of central notions such as scalability, CS students need awareness of commonly used terms that refer to categories or types of parallelism and of communication between processors, such as *task parallelism* (where a computation is accomplished using cooperating functional units or tasks), *data parallelism* (where a given computation is applied in parallel to multiple data sets), *pipeline parallelism* (task parallelism where the output from one task becomes the input for another), *message passing* (in which operations for sending and receiving messages form a basis for communication between tasks), and *shared memory communication* (in which tasks communicate through shared variables or data structures). Note that this collection of terms does not represent a non-overlapping or necessarily a complete decomposition of forms of parallel computation or communication. A more objective classification approach might be taken, such as Flynn's taxonomy [49], which, for example, might describe task parallelism as MIMD (Multiple Instruction, Multiple Data). However, Flynn's decomposition is less commonly helpful in present-day parallel computing, which often carries out entire procedures or programs in parallel, not only individual instructions. We recommend looking to current usage in research and industry when selecting terminology for teaching CS students, while always selecting and formulating those terms appropriately for the intended audience. For example, the so-called SPMD (Single Program or Process, Multiple Data) category of computation [37], a subset of MIMD (Multiple Instruction, Multiple Data) and introduced after Flynn's categories, identifies arguably the most common model of applied parallel computation. SPMD is a useful and enduring notion because it describes the common software strategy in which parallel computing specialists build computational frameworks that domain experts can use to productively program applications. We note that all members of such a project team must know enough about concepts and principles of parallelism to have a conceptual understanding of what they are doing, and that domain experts with stronger parallelism backgrounds can make greater contributions (as can parallelism specialists with domain knowledge).

Another central concept in parallel computing is that of *speedup*, which defines how much faster a parallel solution is than its sequential counterpart. Speedup is measured using the formula:

$$Speedup_C = Time_1 / Time_C$$

where $Time_1$ is the time required by the sequential solution to solve the problem, and $Time_C$ is the time required by the parallel solution to solve the problem using *C* cores. Likewise, the concept of parallel *efficiency* is important:

$$Efficiency_C = Speedup_C / C$$

which indicates how efficiently a given parallel solution uses the hardware's parallel capabilities. Measuring a solution's $Speedup_C$ and $Efficiency_C$ for different values of C provides a way to quantify that solution's scalability with respect to C.

It is not enough merely to know definitions and proper usage of these important terms. Goals for elaboration on and application of these notions in CS curricula appear in subsequent knowledge-area sections.

While scalability issues may arise in almost any context, other concepts relevant to parallelism arise in certain types of scenarios, such as resource management situations. A *race condition* exists when the correct behavior of a program or system depends on timing. For example, if multiple processes both read and write a shared variable, an unfortunate ordering of operations may lead one process to overwrite another process's update of that shared variable, leading to the loss of that update, unless a strategy is devised to avoid such errors. In many CS curricula, race conditions have been introduced in an intermediate or advanced course in Operating Systems or a related topic, in which context a collection of synchronization strategies may be explored for avoiding them (see, for example, [107]). However, the idea that the correct behavior of an algorithm involving a shared variable may depend on timing will arise earlier if one introduces parallel computing earlier in a CS student's coursework, and recognizing the potential for race conditions in parallel code is a valuable skill for CS students. As [80] indicates, the notion of a race condition does not itself require advanced CS training, and is both accessible and even discoverable by beginning students. If the notion of a race condition appears early in a CS curriculum, it can readily be cited in subsequent courses when it relates to topics in parallel, e.g., whenever programming (explicitly or implicitly) with multiple threads and shared memory.

An increasing number of parallel programming environments provide automatic avoidance of race conditions, either by avoiding shared memory (for example, through message passing) or through mutual exclusion synchronization (e.g., "smart" pointer objects with reference counting). However, even programmers with access to such environments need to understand enough about race conditions to choose those features. Given the current rapid and volatile development of parallel computing, CS students will benefit from a broad acquaintance with issues of parallelism, in preparation for the range of computation they may encounter during the span of their careers.

Deadlock is another pervasive issue that may arise in parallel computations. A *deadlock* exists if a set of multiple processes or threads are blocked awaiting events that can only be caused by processes in that same set. Common-sense instances of deadlock in non-computational settings abound, for example, involving vehicles entering opposite ends of a one-way bridge, and general discussions of how to avoid or remedy deadlock in those situations require no particular CS training. Realistic code examples where a typical young CS student can identify deadlock are rare. But an intuitive, common-sense discussion of potential deadlock may beneficially influence the design of parallel algorithms at any level. As with race conditions, a comprehensive consideration of deadlock avoidance and recovery strategies can appear in a more advanced course, as it may already in a curriculum without early parallelism.

The theory of parallel computation is not so widely taught as the theory of sequential computation, nor is a list of topics for theoretical parallel computation so standardized at the undergraduate level as the formal-languages approach in a many Theory of Computation courses. But Amdahl's Law and Gustafson's Law (see, for example, [101] or [17]), two formulas for upper bounds on parallel speedup, are useful for both estimating potential performance improvements and gaining insight into the effects of parallelism in an algorithm. Briefly, Amdahl's Law concerns speedup from parallelizing a sequential algorithm, and Gustafson's Law (also known as Gustafson-Barsis' Law) considers the speedup effects of scaling algorithms that are already parallel. Both formulas are quite accessible to undergraduates, and even a few computations with them can give a student enlightening awareness of the impact of sequential portions of a computation on its parallel speedup.

It would be satisfying to include one or more general theoretical frameworks for parallel computation in an undergraduate CS curriculum. Some candidates exist: the Parallel Random Access Machine (PRAM) model helps define and understand the class of efficiently decidable parallel problems applicable to shared-memory multicore systems[73][46]; the Pi calculus formally models concurrent systems whose configuration may change during a computation, which forms the basis for a wide variety of research efforts and applications [92][93][91]; and tuple spaces, which implement an associative memory with parallel concurrent access, representing an abstraction of distributed shared memory [55].

Other potential topics

The comments above should not be interpreted as prescriptive or encyclopedic, but as evocative for stimulating discussion, as we indicated earlier. In particular, one may well consider other topics to be central in this knowledge area. Examples: fine-grained vs. course-grained parallelism; parallel memory access in the PRAM model; and handling read-write conflicts in concurrent programs.

2.2.2 Software Design

Representative educational outcomes:

Given a problem to solve, CS graduates should be able to:

- Decompose it into sequential and parallel portions,
- Recognize possible parallel approaches that can be used to solve it, and
- Devise and implement an efficient and scalable strategy using a chosen approach.

Central Ideas

The availability of multiple cores brings the potential for improved software performance, but at the cost of increased complexity and difficulty. The advent of multicore processors thus brings a disruptive change to the practice of software design. It also brings a variety of new opportunities to the software designer.

Not all problems should be solved using a parallel computation. For example, computations may have internal dependencies or I/O requirements that necessitate sequential execution. However, multicore hardware may be beneficial even for these sequential computations by using these parallel techniques:

Instance parallelism: Multiple instances of the same computation can be run simultaneously (often launched by a controlling script), each using different inputs and producing different outputs. Given K cores, this approach permits up to K instances of the same problem to be solved in the time required to solve a single instance. Distributed computations such as SETI@Home [106], Folding@Home [50], and similar Berkeley Open Infrastructure for Network Computing [15] projects can be seen as a special case of instance parallelism.

Library parallelism: For some kinds of computations (e.g., matrix algebra), there are libraries that provide parallel versions of commonly needed operations. On a multicore machine, a sequential program may be able to significantly boost its performance by replacing sequential versions of those operations with parallel versions from a library.

Pipeline parallelism: When a sequential algorithm is sufficiently complex, and there are many instances of the problem to solve, a software designer can divide the algorithm into N sequential stages, and spawn a thread or process to perform each step. Ideally, the algorithm is divided so that each stage takes the same length of time to perform, so that the processing load is evenly distributed across the stages, a design aspect called load balancing. The behavior of the thread or process performing *step i* can be described as follows:

```
while (true) {
  if i is the first step {
    data = getInputs();
  } else {
    data = readResultsFromStep(i-1);
  }
```

```
    results = performStep(i, data);

if (i is the last step) {
        output(results);
    } else {
        sendToStep(i+1, results);
    }
}
```

Once the N stages of the pipeline are filled and the first result emerges from the final stage, subsequent results will be produced in time 1/N.

For computations without sequential dependencies, software designers may be able to more directly exploit a multicore processor's parallel capabilities. In particular, software designers who want their software to run faster as more cores are available can try to decompose their software into pieces that will run in parallel on different cores. Two of the ways software can be decomposed into pieces are:

Domain decomposition (data parallelism): the software designer divides the data or problem domain into pieces, and then designs the software to process those pieces in parallel. For example, if the problem is to blur a large MxN image on a machine with K cores, the designer might spawn K threads, divide the image's M rows into K blocks of size M/K, and blur each block in parallel using a different thread.

Functional decomposition (task parallelism): the software designer divides the computation into functional units or tasks, analyzes the dependencies among these tasks, and then designs the software to spawn threads or processes that perform these tasks in parallel (as dependencies allow). For example, if the problem is to model the climate system, and the system's functional pieces are the ocean, the land, the atmosphere, and the water, then the designer might design a task to perform each functional piece, and then execute these tasks in parallel to model the climate. Functional pieces may be further decomposed into sub-tasks, as appropriate.

To run faster as more cores are available, software must be designed so that the number of processes or threads dividing the labor increases with the number of available cores. There are a number of parallel programming patterns that knowledgeable software designers can exploit, including operational patterns like the Map-Reduce and Scatter-Gather, and computational patterns like Pipeline, Embarrassingly Parallel, Geometric, Divide And Conquer, to name just a few [82]. Familiarity with these patterns will ease the work of software designers.

A program that runs faster when given more cores is said to *scale.* Scalability, or potential to scale is a desirable property. Suppose there are two companies P and Q with competing software programs, and that P's program scales but Q's does not. Whenever a system with more cores is released, company P's program will require little if any maintenance to run faster on the new system, giving them a competitive advantage over company Q.

Given a problem, one common pattern in this approach is exhibited by computations based on a parallel for loop:

```
id = getWorkerID();
numWorkers = getNumWorkers();
blockSize = problem.size() / numWorkers;
for (i = id; i < problem.size(); i += blockSize) {
    processPiece(i, problem);
}
```

Another common pattern is the master-worker pattern, in which the same code is run on many processes in parallel:

```
id = getWorkerID();
numWorkers = getNumWorkers();\

if (id == MASTER) {
    combinedResult = divideAmongWorkers(problem);
} else {                    // Worker
    do {
        i = getWorkFromMaster();
        partialResult = processPiece(i, problem);
        sendToMaster(partialResult);
    } while ( i > 0);  // work remains
}
```

One reason for its popularity is the master-worker pattern can be used with either task- or data-parallel computations.

Besides parallelism techniques, CS students need to know enough about parallel software development methods to approach applied problems that require those techniques. The SPMD computational model described in Section 2.2.1 gives rise to the dominant software development method for computing research projects involving parallelism, in which some parallel computing specialists create a computational framework within which a much larger number of domain experts can contribute largely sequential algorithms towards the creation of an application. Google’s use of map-reduce computing [39] provides a widely used example of this approach that can be discussed and (with accessible map-reduce interfaces) used by CS students at all levels, including introductory students. Most research applications developed using SPMD currently use MPI programming (see 2.2.4 below), which is accessible to intermediate and advanced CS students.

Other potential topics

In addition to the above concepts, other important topics that could be considered important enough to include are in the area of quantitative and qualitative measurement of the approach taken. For example, students could analyze their solutions in terms of software metrics related to parallelism such as scalability, performance, reliability, and maintainability. Students could also analyze quantitative experimental measurements of their solutions, or examine the performance of standard benchmarks that use particular patterns [13].

2.2.3 Data Structures and Algorithms

Representative educational outcomes:

Given a problem to solve and a chosen parallel approach to solving it, CS graduates should be able to:

- Choose an appropriate reliable data structure,
- Find appropriate existing parallel algorithms that solve the problem, and
- Devise and implement an efficient and scalable strategy using the algorithm and data structure with that approach.

- Measure the performance changes of algorithms using various numbers of cores
- Compute the speedup and efficiency of parallel algorithms using various numbers of cores

Central Ideas

Data structures have a solid foundation in CS undergraduate programs. When considering certain parallel solutions to problems, students will need to consider whether the data structure they use will be shared among multiple threads/processes. If so, and that structure access is implemented using locks, then it will need to be safe— free of race conditions and deadlock. Libraries of thread-safe data structures exist, such as the concurrent containers in Intel's Threading Building Blocks [65] and Java's java.util.concurrent package ([57] provides an excellent accounting of the complexity involved in implementing such shared structures in Java.)

Because shared access to a common data structure via mutual exclusion locking introduces much complexity in the solutions and they are prone to error, other implementations have devised alternative shared data structure access. Fraser and Harris describe their implementation of multiword compare-and-swap operations or software transactional memory (STM) solutions that do not use locks [51]. Simon Peyton Jones has described the use of STM in Haskell [72], where it is a natural fit. Intel, IBM, and Sun have developed a draft standard for STM in C++, and Intel provides a C++ STM library [66].

Like data structures, algorithms are a foundational topic in undergraduate CS programs. Given the current state of hardware, students will need grounding in parallel algorithmic solutions to problems. Many solutions that heretofore had been part of parallel computing research and found a home in advanced courses should be incorporated into introductory and intermediate courses that feature the study of algorithms. Texts that have taken this approach include [90] and [12].

Many parallel solutions to classic algorithmic problems have been proposed. In the field of numerical computing, vector and matrix operations, as well as Fourier transformations, are good examples. Various parallel algorithms for searching tasks have been proposed; e.g., an algorithm for parallel string matching [123], parallel N-queens problem solvers [111][1], or parallel alpha-beta searches [48], which are relevant in game programming. Sorting is also a relevant field of study, where many parallel versions of quick sort, merge sort, and radix sort, just to name a few, have been described [4][127][79].

Finally, the map-reduce algorithm design paradigm, first found in functional programming, is highly relevant today as a distributed processing technique pioneered by Google [39]. The basic concept of providing a map function to distribute processing and a reduce function to gather and report the results is accessible to undergraduate students. For example, the WebMapReduce software [54][54][53], a simplified interface for the widely used Hadoop open-source implementation of map-reduce, enables CS students at introductory or later levels to develop parallel computations on a cluster by creating the map and reduce functions/methods in a programming language of their choice.

We anticipate that more algorithms and data structures that take advantage of parallelism will appear soon, many arising from computing research and industry.

2.2.4 Software Environments

Representative educational outcomes:

Given a problem to solve and a chosen parallel approach to solving it, CS graduates should be able to:

- Choose an appropriate software library or programming language abstraction for the approach, and
- Devise and implement the solution using that approach.

Central Ideas

Just as CS educators have many choices of languages and software libraries to use for sequential programming instruction, they also have several choices for parallel programming instruction. This is an area under constant development, but good languages and libraries exist that can be used for instructional purposes today. There are several models for parallel programming of cooperating tasks that are commonly used and supported by various languages, language extensions, or systems: shared memory, message passing, actors, and fully distributed.

In the single *shared memory model*, cooperating tasks share information by accessing a shared data space in memory on a single computer. Implementations of this model must ensure that writes to the shared data by multiple tasks are atomic by providing some form of synchronization.

In the *message passing model*, cooperating tasks maintain their own data and share values by communicating them through messages sent and received among those tasks. The message exchange can be synchronous, meaning that a send (or receive) operation causes a task to wait or "block" until another task performs a corresponding receive (or send), or asynchronous, meaning that neither send nor receive operations cause blocking.

The *actor model* is a programming language abstraction built over threads and based on message passing. Developed as a theoretical model to describe parallel computation in the 1970's [60], the actor model has seen a resurgence in the current era of multicore technology. Actors are objects that send and receive messages asynchronously. They can also create new actors.

In *fully distributed models*, the tasks are running on multiple machines connected by some type of interconnection network. An extension of the shared memory concept to distributed systems of computers is called distributed shared memory. Larger-scale fully distributed applications can be realized with systems such as Map-reduce or using message passing. Map-reduce is especially useful when demonstrating how to scale the processing of large datasets, and can be motivated by pointing out Google's use of it to index the web. An open-source implementation of Map-reduce called Hadoop is sufficiently mature and can be installed on clusters of computers or used in the cloud on Amazon Web Services (AWS)[6].

Languages and software libraries support combinations of these models in various ways. Notably, functional programming as a paradigm provides inherent support for parallelism in that it fosters immutable state and closures. Functional languages that have parallel programming capabilities include Concurrent Haskell and Erlang. Concurrent Haskell uses the concept of shared memory between cooperating tasks, and as previously mentioned, avoids the use of locks on the shared memory[59]. Erlang, developed by communications company Ericsson to support distributed applications, uses the message-passing paradigm, rather than shared memory, and employs the actors model[44].

Scala is a relatively new language that was designed with parallel processing hardware in mind and provides an interesting integration of functional and object-oriented features [96]. Scala, which stands for a scalable language, employs an actors-based model. The Scala compiler produces bytecode that will run on a Java Virtual Machine (JVM).

Grand Central Dispatch (GCD), a language extension for C, C++, and Objective-C developed by Apple, provides a programming layer of abstraction over threads. Programmers design 'tasks' reminiscent of closures that are queued up for execution on an underlying dispatch system, which handles the execution of those tasks on threads.

The OpenMP package for C/C++ and Fortran enables programmers to signify sections of code to be run in parallel using shared memory data structures. Portions of serial code can be made to execute in parallel by means of compiler directives called pragmas. Like GCD, work is assigned to threads at runtime.

Intel Threading Building Blocks (TBB) [65] is an object-oriented approach to multi-core programming implemented as a C++ template library. TBB provides a library of object-oriented algorithms, such as loops and sorting, as well as concurrent containers, which are thread-safe data structures such as queues, vectors, and hash maps. Programmers using TBB do not have to be experts on threads– they logically define tasks to be executed in parallel and the TBB run-time library automatically maps that logical definition onto available threads, making efficient use of resources on the machine. Advanced programmers may optionally customize TBB computations at the lower levels of memory allocation, synchronization, etc.

Code adhering to the message passing model can be written using frameworks conforming to the Message Passing Interface (MPI) standard, introduced in 1994. Several implementations of MPI for C, C++, Fortran, and Python are in fairly wide use and well developed. The great majority of high-performance parallel scientific software currently uses MPI, and is often developed using an SPMD-based methodology as described in Section 2.2.2.

A Java library called PJ provides a hybrid solution to parallel programming that blends ideas from OpenMP and MPI in an objected-oriented fashion that naturally fits with the Java language [98]. Programmers can implement solutions on shared memory multicore machines or clusters of computers.

The graphics processing units (GPUs) found in most commodity PCs contain multiple processing units. Though traditionally dedicated to graphical applications, the power of these GPUs can now be used for general-purpose parallel computation using a few different available libraries (when used for more than graphics, a graphics card is known as a general-purpose GPU, or GPGPU). NVIDIA has also developed a programming interface based on C/C++ for programming its GPU hardware architecture, called the Compute Unified Device Architecture (CUDA). There are CUDA wrappers for other languages, such as Python, Java, and Ruby. Microsoft also has a GPGPU API called DirectCompute, for use under Windows Vista and Windows 7.

OpenCL is a new standard, initially developed by Apple, which has been implemented for the GPU hardware of several manufacturers, including NVIDIA and AMD. It is platform agnostic, and lets a program use both the CPU and GPU cores.

Nearly all of these examples of software environments for programming parallelism were developed in academic research projects or industry, and now have become accessible and desirable for CS education. We anticipate rapid growth in this area as parallelism becomes more and more essential in all forms of computing. For example, computing researchers using Sandia Labs' object-oriented Trilinos project [117] provides a base class DistObject for efficiently redistributing an already distributed data object, enabling domain experts to program a small number of conceptually simplified methods that reuse sophisticated parallel techniques through a pattern paradigm. Also, Intel's TBB provides a programmer with "building blocks" in the form of object-oriented algorithms and thread-safe data structures, for constructing efficient and correct parallel applications [43]. CS curricula will especially benefit from software-environment support for effective parallel programming patterns, including new patterns yet to be discovered and developed by academic and industry research.

Finally, we note that these technologies are often used in conjunction with each other in practice. For example, programmers may use MPI together with OpenMP or CUDA on clusters of multi-core or GPU-equipped computers.

2.2.5 Hardware

Representative educational outcomes:

Given a problem to solve and a chosen parallel approach to solving it, CS graduates should be able to:

- Choose appropriate hardware for the approach, and
- Devise and implement the solution using that approach on that hardware.

Central Ideas

Students need to know about machine organization in order to create effective computing applications and systems. It is difficult to understand performance issues without a reasonable mental model of the hardware (a review and comparison of current hardware can be found in [110]). As an example of why this is important, the Sieve of Eratosthenes exercise [101] written in a cache-friendly sequential manner has dramatically better performance than parallelizing in a cache-oblivious manner. Parallelizing the cache-friendly version further increases the speedup. Thus, at least a conceptual understanding of machine organization is necessary to effectively make use of parallelism.

Some of the key hardware organizational schemes relevant to parallel computing today are: multiple instruction, multiple data (MIMD) and its subset, SPMD; single instruction, multiple data (SIMD); and clusters.

Current commodity PCs and servers containing multicore processors are MIMD machines—each core is capable of simultaneously processing data independently. Some of that data can be shared by tasks running on those cores, however. Advanced specialized MIMD machines, such as IBM's BlueGene series, use high-bandwidth connections between racks of many machines in order to support sharing of memory.

The GPGPU cards provide a form of SIMD organization, in which the same instructions are executed by each core simultaneously on separate data. This organization arises naturally for graphics processing, where often the same manipulation needs to be done on every pixel to be displayed.

Clusters of independent machines connected via a network switch, called Beowulf clusters, represent another type of organization commonly used for parallel computing. In cluster computing, there is no concept of shared memory data. Cloud computing is an extension of cluster computing, where cloud vendors enable users to configure their own virtual cluster on remote hardware that the vendors maintain and provide for that purpose.

Exposing students to a variety of hardware topics helps them to develop adequate conceptual models of hardware that informs choices made in the other parallel knowledge areas presented in this section. Simple models are often sufficient to allow the student to begin exploring parallel execution at the beginner level.

Other potential topics

We have intentionally made only general and summary observations in our comments above. Any curriculum will naturally explore specific topics at much greater depth in certain courses than we have indicated, including both parallelism that have become standard in those courses (e.g., instruction-level parallelism) and topics that have recently emerged or re-emerged as important (such as multi-core architectures, interconnection topologies, grid computing).

3. TEACHING AND LEARNING STRATEGIES

Although the list of parallel concepts and knowledge areas we presented in the previous section might seem daunting, we contend that there are straightforward mappings from existing curricula to updated ones that include these topics. To help students learn to "think in parallel," we assert that a spiral and experiential approach to the topics is desirable. Although we are advocating a particular approach, we recognize that each institution has its own unique curricular challenges and opportunities. We respect and embrace that diversity. In the following we attempt to identify topics that are likely to be part of many curricula and show how particular ideas in parallelism might be included with these likely existing topics.

3.1 Overarching Approach

To help our students learn to think in parallel, we advocate a spiral approach to these topics, introducing ideas at a basic (though "intellectually honest") level early, then revisiting the ideas with greater depth and formality later in a curriculum [23][24]. This approach encourages a focus on fundamental ideas. As Schwill wrote, "A subject is more comprehensible if students and teachers grasp its fundamental principles. Fundamental ideas condense information by organizing incoherent details into a linking structure which will be kept in mind for a longer time. Details can be reconstructed from this structure more easily" [105].

Along with a spiral approach, the hands-on practice of experiential learning will allow students to test their own conceptual frameworks. We are not advocating the teaching of tools over concepts, rather we are advocating experiential learning so students can engage with concepts and test their own conceptual models. Educators and students should expect changes to tools and techniques. By ensuring that our students are well grounded in the concepts, we prepare them to adapt to these changes. Many different existing tools and techniques can be used to introduce and expand upon the concepts we have outlined in the previous section. We will provide some examples below.

We believe that the key to this process is in problem selection: sprinkle problems through the curriculum that benefit from parallelism, but whose solutions introduce new issues, the resolution of which expands a student's understanding step by step. By repeatedly exposing students to topics through the spiral approach, we can help students to recognize the underlying concepts, not just the specific technologies. In doing so, we hope to optimally prepare them for whatever emerges in the future.

The following is an illustrative example using particular technologies. The basic philosophy is to start gently in an introductory course, with students devising sequential solutions to embarrassingly parallel problems like matrix addition or simple image processing tasks. In the next course, instructors using C++ for example, could then use OpenMP to have students solve the same problems with parallel solutions. We have found that OpenMP is easy enough to use that a simple introduction to multicore technology and threads is enough to get the students up and running, including those who are not CS majors. The key is to motivate the material through a hands-on exercise that lets them directly experience the benefit of parallelism, visually if possible.

As an example, an exercise involving a standard computation might increase the size of a data structure (i.e., matrix or image) enough that the students become impatient waiting for a sequential solution to complete. Given the speeds of modern computing equipment, young students often become accustomed to their computations completing almost immediately, so they may perceive running times of only a few seconds as being unexpectedly long. Thus, if a student's parallel implementation completes in a fraction of the time required for a sequential computation, that student directly *experiences* the performance speedup. Several of us have found that a hands-on experience of parallelism speedup becomes a personal turning point for some students, motivating them to learn more about modern parallel computing practices and equipment, sometimes even leading non-majors to take more CS courses.

In subsequent courses like operating systems, we can then give problems whose multithreaded solutions introduce race conditions, deadlocks, etc., and so motivate the use of synchronization constructs. Likewise, a course like programming languages can introduce the benefits of functional languages (e.g., immutability or STM) for parallelism, and compare the concurrency mechanisms of different languages. It is also conceivable to cover actors as a natural way of implementing parallel software when object-oriented abstraction is used. Advanced elective courses can delve even further, because students have seen and understand the concept of using parallel solutions to solve real problems.

3.2 Adapting Existing Curricula

There are many natural places in most curricula where parallel topics can be introduced. Introducing parallelism in this way results in broad, incremental, but perhaps not deep, changes to courses. However, we anticipate that the cumulative effects of these incremental changes will be deep changes in students' abilities to think in parallel.

3.2.1 Introductory Level

The spiral approach argues that even introductory CS students should be introduced to parallelism beginning early in their careers, including a significant number of students who do not continue in CS but who will use computing in their work.

There are already several published examples of faculty successfully introducing parallelism topics early in the curriculum:

- When introducing event-based programming or graphics, it is natural to use multiple threads of execution. By focusing on race-free problems, Kim Bruce and his colleagues Andrea Danyluk and Thomas Murtagh have successfully used this approach for many years at Williams College and Pomona College [22].
- Daniel Ernst and Daniel Stevenson (University of Wisconsin/Eau Claire, U.S.) argue for introducing parallel computing and concurrency using applications such as image processing and encryption cracking [45].
- The WebMapReduce software [54] produced at St. Olaf College and associated teaching material [97] enables introductory students to program with Google-style distributed map-reduce computing [39] with Hadoop [7], using Java, C++, Python or Scheme to create mapper and reducer functions/methods.

All these examples involve experiences that connect a student's programming with appealing applications having external importance. In Bruce et al's work, students write parallel programs using "active objects", simple library classes that encapsulate threads and provide threading behavior through simple inheritance. In Ernst and Stevenson's examples, students write concurrent programs using Java threads directly. In WebMapReduce, students write code that is executed in parallel by a fault-tolerant, data-parallel system that is capable of handling large-scale data sets. In all these cases, accompanying discussion can relate the activities to broader issues in parallelism.

Another way to introduce parallel thinking in introductory courses is through the use of "Computer Science Unplugged" sorts of examples [30]. For example, one of us uses a simple survey where he asks students, "How do you multitask? Please describe below how you use your time while doing multiple things within the same time period." He collects this information and categorizes it according to a particular approach to parallel programming, say pipelining or multi-threading. Discussing these results in class provides an opportunity both to demonstrate that parallelism is a common feature of daily life and to foreshadow later classes in the curriculum. Because the examples are drawn from the students' own experience, students are more likely to see engage in the discussion.

Lewendowski and colleagues have recently shown that students in the very beginning of introductory CS courses, when given a problem posed in English prose, are able to grasp the problems of concurrent access to shared data by multiple tasks [80]. The problem they posed was one of people buying concert tickets from multiple sales clerks at a bank of ticket windows. The sales clerks represent multiple parallel tasks, and the information they share is which tickets have been sold and which are still available. This problem could be posed in an introductory course, and students could be asked to formulate solutions. Those solutions could be discussed, thereby giving students their first taste of race conditions and how to handle them, without getting into the details of hardware design that have traditionally been needed to understand them in advanced courses.

3.2.2 Concepts-first Curricula

Some institutions prefer curricula that focus on concepts first, sometimes to the exclusion of studying computers as physical machines early in students' careers. These programs might choose to introduce abstract machines, such as Concurrent Sequential Processes (CSP) [61] or the Pi Calculus [92][93][91], early in the curriculum, deferring hardware parallelism and software environments to later in the curriculum.

3.2.3 Data Structures

In courses that introduce or use common data structures, there can be a natural transition from the study of individual sequential access to those structures to considering issues of safe concurrent access to those entities, whether through appropriate use of synchronization structures or through lockless data structures. For example, one of us has used an exercise where students initially build a sequential, single-threaded web crawler in Java, storing links found in traditional list, map and queue data structures. Following that, we introduce the use of threads and have the students build a multi-threaded crawler using the corresponding thread-safe data structures found in the *java.util.concurrent* library.

3.2.4 Software Design

In intermediate courses on software design and development, the ability to identify and take advantage of parallelism becomes an essential skill.

In a course that teaches software patterns, an instructor might choose to introduce a pattern or two for parallel processing. There are many parallel patterns identified already [82], and more work on identifying such patterns is on-going [64][74]. By introducing a pattern or two in an existing course, instructors can keep parallelism in front of their students. Some institutions might consider using a broader selection of the parallel patterns in a later course.

Many traditional software patterns also provide opportunities to discuss parallelism. For example, when discussing model-view-controller [78][100], an instructor might discuss how long-running events in the controller may need to be executed in parallel to allow the view portion of the program to remain responsive to user input. Similar observations about potential uses of parallelism arise in other traditional patterns such as adapter, proxy, or observer [53].

A course that discusses software architecture might also provide a profitable place to introduce some parallelism. For example, when discussing client-server architectures, an instructor can discuss the problem of scaling the system with the number of clients. The server will have to be multi-threaded (whether manually or through the underlying implementation platform) to be able to serve more than one client at a time. Students can readily grasp this need for parallelism.

Courses that include team projects provide another opportunity for Unplugged exercises. An instructor can prompt interesting discussion with questions like, "How are the members working on a team project an example of parallelism?", or "How is a team employing an agile methodology an example of parallelism?"

3.2.5 Algorithm Analysis

Courses that discuss algorithm analysis might include elements of the distributed paradigm, notions of consensus and election, and issues of fault tolerance. These topics remain relevant in the multi-core age; indeed, early processors with many general-purpose cores (such as Intel's recently released 48-core research chip [109][62] and their experimental 80-core network-on-chip [121][83]) arguably behave like distributed systems in many respects. Topics such as assessment of parallel speedup (e.g., Amdahl's Law), data scalability, categories of parallel computing (e.g., data parallelism vs. task parallelism), relative performance of various thread computing models rise in importance in the presence of large-scale multi-core computing.

In a course that covers sorting algorithms, an instructor could introduce parallel merge sort, for example. This is just a small step beyond traditional merge sort and does not require substantial additional time. Introducing an algorithm like parallel merge sort helps students to begin to consider the parallelizability of an algorithm as an important facet in choosing the algorithm to use in a particular circumstance.

3.2.6 Programming Language Concepts

Many people think that some previously underused programming language features, such as functional programming, actors [60][3], and software transactional memory (STM) [76][51] will become more important in the age of multi-core [21]. Independent of this, we seem to be in an era where practitioners are expected to be fluent in a variety of languages. For example, many application development teams use software frameworks such as Thrift ([8], originally developed at Facebook.com) to combine services built from many different languages. For these reasons, programming language concepts are rising in importance. Institutions that have de-emphasized programming language concepts in the past my wish to reevaluate those decisions. Institutions that have maintained programming language concepts in their curricula should consider introducing parallelism concepts in the following ways.

- Institutions that approach programming languages using a breadth-first, variety-of-languages approach may wish to consider using a language like Clojure [58][27] or Haskell [59], which include support for STM. Scala [96][116] and Erlang [44] include support of the actor model. Fortress [5] is being specifically designed with parallelism in mind; for example, *for* loops are parallel by default.
- If an institution chooses to use a narrower set of languages, many of these concepts can be approached using libraries. For example, the Kilim [112] and Candygram [25] libraries provide support for implementing actors in Java and Python respectively. Duece for Java [40] and Axon for Python [10] would let students experiment with STM. The JCSP libraries implement CSP in Java [29].
- Institutions who approach programming languages using a more theoretical approach might consider introducing parallelism using models like CSP [61] or the Pi Calculus [92][93][91].
- Institutions who approach programming languages by constructing a series of interpreters [52] should consider incorporating concurrency in the interpreters [120].

3.2.7 Operating Systems, Computer Architecture and Organization

If students have been begun to develop a conceptual model of parallelism in introductory courses using examples such as we have described above, then this will enable deeper study and understanding of concepts such as instruction level parallelism, hyperthreading, and multicore architecture, which are traditionally studied in courses involving computer architecture and organization. The introduction of parallelism earlier may free up time in such courses to cover other topics.

While parallelism is a traditional topic in operating system courses, other courses dealing with systems software can also benefit from curricula extended with concurrency and multithreading aspects. For instance, courses that address programming language implementations in the form of virtual machines could cover the implementation of parallel language features such as actors in terms of lower-level features such as threads and locking. As a concrete example, the way the Erlang VM implements the language's lightweight actor model can be studied in depth.

3.2.8 Advanced Electives

The integration of parallelism topics throughout the curriculum still leaves room for advanced electives in parallelism. In fact, these electives will be able to explore advanced topics even more deeply when students arrive in them with basic concepts already in place. For example, if students have used a system such as WebMapReduce to implement solutions to basic data-parallel problems earlier in their studies, then a special course in parallel programming or distributed systems could delve deeply into the use of Hadoop to solve sophisticated problems, such as creating a PageRank [19] type of index of all of Wikipedia. As another example, if students have already used OpenMP to implement data parallel solutions, they should more easily be able to consider message-passing solutions to problems requiring sophisticated task-parallel solutions, using any of a variety of tools, such as MPI or TBB. Rivoire describes an advanced undergraduate course that integrates several of these technologies [104].

3.3 Language Issues

The increasing importance of parallelism affects the choice of programming languages used in teaching. Going forward, educators will have to consider the support offered by a language for various parallel development concepts. Different institutions will choose to make this choice in different ways.

Teaching languages like Alice [33] or Scratch [103] already offer significant support for basic parallelism. In particular:

Alice provides four constructs for specifying sequential or parallel control:

A `DoInOrder {}` block that executes the statements in the block sequentially

A `DoTogether {}` block that executes all the statements in the block simultaneously

A `ForEachInOrder {}` loop that sequentially performs the statements in the block, for each item in a list sequentially

A `ForAllTogether {}` loop that sequentially performs the statements in the block, for all items in a list simultaneously

These constructs make it remarkably easy to simulate real-world simultaneous behaviors in Alice. For example, to simulate a basketball referee raising both arms to indicate a successful 3-point basket, one might program:

```
DoTogether {
      referee.leftArm.turn(forward, 180);
      referee.rightArm.turn(forward, 180);
}
```

Scratch provides a simple event-based mechanism for specifying parallel behavior. For any event *e*, any Scratch *sprite* (i.e., animate-able object) or the *stage* can define handlers for *e*. When event *e* occurs, all handlers for *e* are executed simultaneously.

When shown their use, young students find these constructs remarkably intuitive and easy to use. Scratch and Alice are thus engaging ways to introduce students to parallel thinking.

Although sequential computing has dominated in CS education since the field's inception, many aspects of parallel computation are actually quite natural and accessible, even to pre-college students. Beginning CS students are capable not only of understanding parallelism concepts and programming examples, but also of discovering issues such as those surrounding concurrent access to shared resources [118][26][80]. For many problems within computing and beyond, parallelism provides the most natural solution strategies, and we encourage identifying and acting on the natural opportunities for parallelism in CS courses.

Python has been growing in importance as an early teaching language. The current standard Python implementations use a "global interpreter lock" to ensure that multiple threads running in the interpreter mutate the interpreter's state in a way that is consistent with a sequential execution of the thread's code [56]. In essence, this means that multi-threaded Python code cannot take full advantage of multi-core processors. Other Python implementations, such as IronPython [69] and Jython [115] do not have a global interpreter lock, and so do not suffer this relative disadvantage. However, these implementations are not necessarily compatible with the standard APIs and libraries that an instructor might wish to use in a course.

4. SUPPORT: A SHARING COMMUNITY OF EDUCATORS

4.1 Online Community

Effective and productive sharing in the environment of a welcoming and lively community of peers can make a tremendous difference in supporting faculty during this time of transition toward the parallel future of computing. Given the great diversity of faculty experience and interest, and the widely varying nature of curricular needs, no single source of support will be sufficient for all situations. However, a broad community of educators seeking to bring about systemic change can profoundly assist others who wish to enhance their backgrounds in parallel computing by thoughtfully creating a variety of strategies for training and supporting each other.

To support broad based changes to curricula such as we have proposed in a timely manner, we will need to share material and ideas by electronic means and form a community. Though several repositories currently exist offering access to parallel and more general computing content with various levels of metadata and search capabilities, these resources are still underutilized [94]. Moreover, most of the enhancements to courses that we suggest in this report are not yet placed in any easily accessible location that will facilitate fast adoption. In this section we consider some existing web sites for sharing CS educational materials, discuss barriers to forming a true community of educators, and suggest ways that we might overcome them.

Several efforts towards sharing CS educational materials are currently underway.

- Ensemble (http://www.computingportal.org/), funded by the National Science Foundation, is a "Pathway project" of the National Science Digital Library(NSDL). Aimed at the entire computing education community, this project seeks to encourage contribution, use, and review of a broad range of CS educational materials. There is currently no material for parallel computing on this site, but a community devoted to introducing parallel concepts throughout the curriculum has been formed there.
- Another part of the NSDL is the CITIDEL project (Computing and Information Technology Interactive Digital Library, http://www.citidel.org/), which serves as a repository for computer science education materials. Its 'computing materials community' contains several collections, organized by relatively standard course subject areas, such as computer graphics, networks, and algorithms. Educators can join and contribute materials such as lecture slides, exams, and papers describing classroom techniques. Users can browse by the communities and collections, or by author or date. In an attempt to provide controlled vocabularies, users can also browse by subjects, which are organized by the ACM Computing Curricula recommendations.

There are some sites directed more specifically at the parallel computing education community.

- Intel Academic Community (IAC) provides an open platform for sharing parallelism courseware components [2]. The components are organized using the controlled vocabulary of the ACM Computing Curricula Detailed Body of Knowledge ([68], p.36). The platform itself is based on Moodle technology, widely understood and adopted by academia. The open platform allows for both downloading and linking to exiting course material as well as the uploading of new items. Intel has instituted both a ranking and reviewing capability for broad community input as well as a system of expert reviews.
- OpenSparc sponsors a wiki called "Sharing Teaching Material for Concurrent Computing" [32], which is designed with collaboration in mind. Instructors provide course materials under the Creative Commons License that are freely accessible by other educators.
- The CUDAZone site, sponsored by NVIDIA, serves as a repository for code examples in a variety of application areas [36].

These resources are a start towards making materials available to educators for searching and browsing. But courseware repositories alone will not be enough to meet the goal of forming a community of educators who are trying to change their curriculum quickly in incremental ways. We need to be communicating by sharing ideas about what works and what doesn't, discussing learning objectives, comparing classroom techniques for teaching various new topics, and providing peer reviews of shared materials. ome sites are already seeking these objectives. Notably, the Intel Academic Community website hosts relevant blogs and forums, and produces video content including the weekly "Teach Parallel" broadcasts [114]. Building from this significant beginning, website providers must discover and incorporate features and content that are ever more accessible, convenient, and valuable for this purpose, and CS educators must begin incorporating the resulting rich online resources into their course preparation practices and teaching, in order for the vital community we envision to emerge.

We face several challenges to establishing an active and vibrant on-line community for parallelism in the CS curriculum:

- The episodic nature of course development—busy professors may only access these resources when they need specific materials, or when they have completed a project and are ready to share their results.
- A lack of incentive—professors are typically worried about completing tasks that are valued for promotion and tenure (research, teaching, and service to profession); it is not clear that participation in such a community will be valued by those outside the community as viable service.
- A small community size—popular sites with ratings and comments and blog posting rely on strength on numbers to develop a strong community of users; our community of CS educators is likely to remain relatively small.
- Shared content may not be easily adapted by others—to be widely shared, some content will ideally to be either platform independent, or easily ported from one environment to another. This is not always an easy prospect for busy professors.
- Technology changes quickly—some materials may lose applicability and relevance as hardware and software evolve.

We see some important features to add to online communities to facilitate sharing. The ability to find relevant and appropriate content is as important as the content itself. There are many salient features that should be included as metadata associated with the shared content to support its easy accessibility by course developers. These will provide attributes for searching such as its level of difficulty, learning objectives, platform requirements, or language used.

Although user rating and comments offer value to consumers, the benefit of peer review is more highly valued and sometimes required in academia. However, requiring peer review prior to posting may slow the sharing process, so it is not something we would advocate initially. Traditional online rating systems, such as 1-5 stars, may also not be appropriate for or widely used by our community.

The success of sharing of course content inevitably falls upon the participants involved. Besides developing shareable content, academic community members need to aware of the mechanisms available for sharing, and then actually take part in the sharing process, including posting of sharable content, downloading and using what is available, and providing feedback to the community. It is equally important to enrich the breadth of content offerings by capturing the work of community members as they port student exercises to other platforms or update course materials to accommodate revisions in OS and related tools. To make this easier and to test materials, remote systems are available that allow academicians to access manycore systems. The Elastic Compute Cloud (EC2) from Amazon Web Services (AWS) provides inexpensive access to virtual instances of machines with preconfigured numbers of “virtual cores” [6]. Intel’s Many Core Testing Lab (MTL), which offers community access to advanced systems expressly for the purpose of creating and testing content as well as use by the community as a teaching platform [67]. Such efforts must continue to be expanded in order to build a critical mass of available teaching content.

These expectations for testing and sharing materials are somewhat at odds with the episodic nature of course development in academia and with the value we receive from participating. Some of us however, will need to take part in order to meet the goal of changing our curriculum to keep pace with the change in hardware.

Besides offering an effective sharing environment such as a repository, it is necessary develop a community of participants conducting a variety of supporting activities including but not limited to posting and using content, as well as commenting on posted content. We need to induce repeat visits independent of immediate courseware development needs. Likewise other mechanisms for informing members of new and updated content, such as RSS feeds, can provide similar value for more passive members. Nevertheless the success of such efforts is still dependent on available time and motivation of often-overworked professors to participate in the process. Hopefully some of us will take up the charge to be proactive.

By stating these objectives for online resources, we do not imply their absence from existing sites. For example, many sites include forms of search and metadata that help an instructor identify materials suitable for a particular situation. As another example, Intel Corporation has chosen to assign a high priority to online support for the international community of computing educators teaching parallelism, as members of that community create materials and curricular innovations appropriate for their own diverse academic situations. This effort shows leadership in both energetic building of online supports for community and in respectfully encouraging CS educators to develop parallel teaching materials. Yet much work remains on these objectives for discovering and implementing ever more effective support mechanisms, in order to engage and serve the emerging heterogeneous populace of CS educators who will soon begin teaching parallelism in new curricular contexts.

Developing an on line community can be likened to throwing a party. You can offer all kinds of party favors to get things started, but your success ultimately depends on keeping those who arrive early around long enough to develop a critical mass wherein it is the quality of the participants and their activities that comprise the value proposition that keeps the party going. Not only should your party be broadly advertised, the benefits of attending need to be well understood. Once the parallel CS educator community decides on appropriate meeting spaces online, we need to both advertise and participate in order to be effective.

4.2 Workshops

Community development can be further reinforced through live, face-to-face interaction between both active and prospective members. Workshops (which are an established feature associated with many technical and academic conferences), can offer not only technical training in parallel programming but also perhaps more importantly focus on curriculum development, including case studies documenting professors’ real accounts of curriculum development, outlining obstacles encountered and tactics for overcoming them (for example, [31]). Workshops have proven to encourage community development and contribution of materials to repositories, as described in [70]. It now is feasible to create remote videoconferencing workshops [95]. Not only will these activities invigorate our community at a grass roots level, but also provide an opportunity to cross-pollinate parallel curriculum development across a variety of disciplines within the scopes of the various targeted conferences.

5. BRINGING ABOUT SYSTEMIC CHANGE

5.1 An approach to change

As the foregoing sections indicate, we view parallelism as a topic area that would ideally appear at many locations in a CS curriculum, and at the elementary, intermediate, and advanced levels. *We recommend a spiral approach to exploring fundamental concepts of parallelism, with enduring principles always reinforced by "hands-on" experiential learning.* Several factors convince us of this position.

- Since multi-core computing will soon have a profound effect on nearly all forms of computation, even non-major students who may only study CS for a term or two will benefit from exposure to fundamental and enduring parallelism concepts, however they may use computing in their future academic and employment careers.
- As every advertising agency knows, frequent periodic exposure to information helps a hearer to understand and retain that information, and also implicitly conveys a sense of that information's significance. Likewise, frequent appearance of concepts and practice in parallelism throughout a CS curriculum will likely prepare students better for the future of ubiquitous parallel computing they will face than a more concentrated exposure localized within a single course.
- The scientific study of learning and education research support teaching strategies such as hands-on "active learning" ([18][88][85][84][87][86]) and a spiral approach to presenting and learning deep and substantial concepts over time ([77][23][24][41]). For example, we note that the spiral approach is considered a core learning pattern in the field of CS Didactics, found in several European countries, which concerns the scientific scholarship of learning in our discipline [105][113][20].

These strategies for teaching parallelism suggest a systemic change in CS curricula, in which strategically selected topics in parallelism appear frequently in the contexts of numerous CS courses. While we do not claim to know how to untie the Gordian Knot of transformative change, we suggest three guidelines that may help the community to succeed in injecting more parallelism into CS curricula.

1. *Incremental change*. Cyclically making strategic small modifications helps to create manageable change.
2. *Assessment*. Assessing results helps to identify and quantify the effectiveness of change.
3. *Proceeding in and from community*. A collaborative, open, and supportive community provides an excellent environment for fostering change among the widest set of stakeholders.

Some examples may shed light on how these guidelines can function together to encourage systemic change. The field of information security was represented in the 2001 version of the ACM-IEEE/CS curricular guidelines in the titles of only two knowledge units, one required (3 hours) and one elective [71]. But the rising importance of the field in present-day computing and the energetic incremental activity of the information security community led the 2008 revision of those curriculum guidelines to include six knowledge units with titles involving security, three of which are required (12 hours total) [68]. Note that the guidelines include assessment cues (such as the number of required hours and learning objectives for each knowledge unit) that help to identify and quantify these curricular changes.

We believe that an incremental, assessable, and community-based approach will also serve the CS education community as it introduces more parallel computing concepts in undergraduate CS curricula

5.2 Some challenges

We comment below on some of the challenges that the CS education community may face as it seeks to expand the presence of parallelism in CS program offerings.

5.2.1 CS curricula are already full

The most obvious concern about bringing more parallelism into CS curricula is the challenge of adding topics in parallelism to courses and academic programs that are already overstuffed with content. We have made no attempt to recount a thorough justification for teaching more parallelism to CS students, but instead we have assumed that premise and focused on strategies for instituting that change in CS curricula.

However, we note that leading corporations in information technology and top research groups have already made a strong case for this curricular transformation. To quote John Shalf of Lawrence Berkeley National Laboratories in connection with the recent announcement of Intel 32- and 50-core CPU products to be delivered by 2011, *"[T]he most important thing moving forward is that everyone is now affected by parallelism. So the ideas have moved from being the interest of a handful of academics and supercomputing advocates to a mainstream problem that is important to the broader computing industry – the likes of Microsoft and Intel. This means it is even more imperative that we train future computer scientists to solve problems using parallelism from the get-go. The transformation of our educational system will be as big and disruptive as the changes to our software environment. So we'd better start now"* [47].

Given the urgent need to bring more parallelism into the CS curriculum, some traditional course content may be displaced for at least the short term. Over time, we anticipate that the CS education community will discover cost-neutral substitutions and rearrangements of topics within courses that will allow concepts of parallelism to be taught with little or no sacrifice of other course topics. In fact, several examples of efficiently introducing parallelism appear in Section 3. As research and industry develop new parallel programming algorithms, data structures, and environments, the creative CS education community will surely discover better and better curricular optimizations incorporating them.

5.2.2 The urgency of this change

CS curricula do change over time. For example, the now universal acceptance of object-oriented programming in undergraduate CS education shows that systemic curriculum change is possible in our discipline. However, that transformation took about 20 years to accomplish, which is far more time than we have to produce CS graduates who are competitive in parallel computing, given the rapid expansion of programmer-accessible parallelism in today's computer hardware.

Fortunately, incremental change is an effective strategy that initially requires only a small allocation of time and equipment.

Students who learn a few strategically chosen lessons in parallelism now will graduate to academic and employment careers with useful lessons and some valuable insights. If subsequent students learn more parallelism, the accumulation of modest steps in parallel computation can soon lead to deeper education in the principles of parallelism, strengthened by broad experiential learning activities.

5.2.3 Faculty challenges

Teaching parallelism at all levels of a CS curriculum, and in courses that have not traditionally been associated with parallelism, places significant demands on the faculty teaching those courses. Up to now, parallelism has been a specialty of a small proportion of a CS faculty. The rapidly changing nature of all CS subfields has always placed a substantial burden on CS faculty in keeping up with their academic interests and responsibilities. Each CS professor must make his or her own cost-to-benefit calculation about how much time to invest in each new technology or disciplinary development, including the increasingly prominent developments involving parallelism. The timely success of a systemic change towards teaching more parallelism in the CS curriculum will hinge on making it as easy as possible for professors who are non-specialists in parallelism to develop competence in the new topics they now need for their courses, and to obtain the course materials and parallel computing infrastructure support required to teach those new topics.

As indicated at the beginning of Section 4, an intentional supportive community of peers is key to helping faculty making this transition toward the parallel future of computing. Such a community will bring parallelism specialists and non-specialists together, and will also connect persons with similar professional interests outside of parallelism, for example, instructors of a given type of course or persons with common academic interests. Some example strategies include: technical workshops, whether on-site or remotely accessed; workshops on teaching and learning strategies, presented by educators, which may focus on particular courses common among many institutions' CS curricula, or particular teaching techniques, etc.; formation of mentoring networks, or other relationship-based support strategies; and the creation and maintenance of effective online resources for locating, finding, sharing, and developing resource materials, assignments, and pedagogical systems that support the teaching of parallelism. See Section 4 for further elaboration on this community-based approach.

6. SUMMARY OF RECOMMENDATIONS AND FUTURE WORK

The following recommendations summarize the results of our study of strategies for injecting parallelism into CS curricula.

1. Teach a broad collection of enduring principles of parallelism through hands-on experiential learning, in order to prepare students best for the volatile future of parallel computation while reinforcing concepts through active learning.
2. Let research findings and industrial developments inform the modifications of courses and curricula to include more parallelism. We present a suggested framework for a body of knowledge for parallel computing that derives from the recommendations of researchers in academia and industry.
3. Teach parallelism early and often, at all levels of undergraduate CS curricula. Both CS majors and non-majors need background in parallelism, and the material merits ongoing vertical study during a major's undergraduate career.
4. Use a spiral approach for presenting substantial concepts and issues in parallelism, as recommended by the European field of CS Didactics, and incorporate other results of the sciences of teaching and learning when introducing concepts and practices of parallelism in new locations in CS courses and curricula.
5. Select concepts and applications strategically when introducing parallelism in elementary courses and other non-traditional locations in CS curricula. The depth and complexity of parallel computing is usually less important than accessibility and its perceived significance for beginners.
6. Develop an open, inviting community of CS professors, administrators, and staff who are seeking to bring more parallelism into the CS curriculum. Include both specialists and non-specialists in parallelism, and encourage collaboration, training, and mentoring relationships among all community members.
7. Create effective online resources to support that community of CS academics seeking parallelism in CS courses and curricula. Some elements of such online resources include:
   a) community review of online materials, with appropriate incentives and academically recognized rewards;
   b) creative features for search, sharing, feedback, and ongoing discussion that make sites much more than mere repositories; and
   c) open cross-listing and linking between such sites, digital libraries such as NSDL, and other resources in order to provide wide, convenient access and to encourage a sharing and supportive community.
8. Approach the shift towards more parallelism in CS courses and curricula strategically, beginning with incremental change and proceeding in and from community.

We also have several suggestions to the international CS educations for future work that would build on our findings.

- Present experiences about efforts to implement these ideas in various institutional types and curricular settings, at national and local meetings. We note that assessed work will be most persuasive and edifying for the community, and that well-analyzed failures often yield as least as many lessons as smooth success stories.
- Collaborate with other faculty on this challenge of parallelism. Collaboration of peers on new initiatives can reap the strengths and minimize the weaknesses of all. Mentoring relationships between newcomers and more experienced persons typically benefit both parties, both professionally and personally, as well as in the subject at hand. The injection of parallelism into CS curricula will require rapid and effective systemic change, and an open spirit of collaboration is the primary ingredient in forming the community of CS educators needed to support that transformation.
- Develop opportunities for workshops and other gatherings of CS educators, for purposes such as training in associated technologies, and sharing strategies and vehicles for teaching concepts of parallelism. We note that workshop sessions often accompany professional meetings such as SC (SuperComputing), SPLASH (formerly OOPSLA), SIGCSE, and others, and some of these are long established, well

supported, and very effective. However, many additional venues and various types of gatherings and workshops will be needed to reach the great variety of professional backgrounds, institutional settings, and curriculum requirements that must be served.

- Continue to explore strategic aspects of bringing parallelism into the CS curriculum. Some of the aspects identified in this report deserve deeper consideration. Two immediate needs in preparation for the coming transformation are the expedited development of effective teaching and learning strategies for parallel computing content, and the collaborative establishment of effective means of sharing among a community of CS educators who will gather around teaching parallelism. Another substantial issue worth studying that is not a focus of the present report is tenure and promotion rewards structure for producing and sharing high-quality online materials related to teaching and learning parallelism.

7. ACKNOWLEDGMENTS

Most of the content of this report was developed during an electronic discussion that took place during the two months prior to our intensive writing meeting in Ankara. Our discussion group consisted of the authors and an expert advisory panel who freely and energetically informed and challenged us, greatly influencing and benefiting our work. The members of our advisory panel were: Clay Breshears, Intel Corporation; Daniel Ernst, University of Wisconsin—Eau Claire; Gregory Gagne, Westminster College and coauthor of [107]; Michael Heroux, Sandia National Labs and St. John’s University; Jeanne Narum, Project Kaleidoscope; and Matthew Wolf, Georgia Tech and Oak Ridge National Labs.

8. REFERENCES

[1] Abramson, B. and Yung, M. 1989. Divide and conquer under global constraints: A solution to the N-queens problem. *Journal of Parallel and Distributed Computing*. 6, 3 (Jun. 1989), 649-662.

[2] Academic - Intel® Software Network. *http://software.intel.com/en-us/academic/?cid=cim:ggl|academic_us_brand|ks112A7|s*. Accessed: 08-06-2010.

[3] Agha, G. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press.

[4] Akl, S.G. 1990. *Parallel Sorting Algorithms*. Academic Press, Inc.

[5] Allen, E., Chase, D. et al. 2005. The Fortress language specification.

[6] Amazon Elastic Compute Cloud (Amazon EC2). *http://aws.amazon.com/ec2/*. Accessed: 07-28-2010.

[7] Apache Hadoop Project. *http://hadoop.apache.org/*. Accessed: 06-28-2010.

[8] Apache Thrift. *http://incubator.apache.org/thrift/*. Accessed: 08-02-2010.

[9] Asanovic, K., Bodik, R. et al. 2009. A view of the parallel computing landscape. *Commun. ACM*. 52, 10 (2009), 56-67.

[10] Axon: STM. *http://www.kamaelia.org/STM*. Accessed: 08-02-2010.

[11] Benson, D.A., Karsch-Mizrachi, I. et al. 2010. GenBank. *Nucl. Acids Res.* 38, suppl_1 (Jan. 2010), D46-51.

[12] Berman, K.A. and Paul, J.L. 2004. *Algorithms: Sequential, Parallel, and Distributed*. Course Technology.

[13] Bienia, C., Kumar, S. et al. 2008. The PARSEC benchmark suite: characterization and architectural implications. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (Toronto, Ontario, Canada, 2008), 72-81.

[14] Bisciglia, C. and Kimball, A. 2008. Getting started with cluster computing for undergrads. Vendor Session. *The 39th ACM technical symposium on Computer science education* (2008).

[15] BOINC. *http://boinc.berkeley.edu/*. Accessed: 08-06-2010.

[16] Bransford, J. and National Research Council (U.S.).;National Research Council (U.S.). 2000. *How people learn : brain, mind, experience, and school*. National Academy Press.

[17] Breshears, C. 2009. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media.

[18] Briggs, T. 2005. Techniques for active learning in CS courses. *J. Comput. Small Coll.* 21, 2 (2005), 156-165.

[19] Brin, S. and Page, L. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*. 30, 1-7 (Apr. 1998), 107-117.

[20] Brinda, T. and Schubert, S. 2001. Didactic system for object-oriented modelling. *Proceedings of the IFIP TC3 Seventh IFIP World Conference on Networking the Learner: Computers in Education* (2001), 473–482.

[21] Bruce, K. and Freund, S.N. 2008. Programming languages as part of core computer science. *SIGPLAN Not.* 43, 11 (2008), 50-54.

[22] Bruce, K.B., Danyluk, A. et al. 2010. Introducing concurrency in CS 1. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 224-228.

[23] Bruner, J. 1974. *Toward a Theory of Instruction*. Belknap Press of Harvard University Press.

[24] Bruner, J. 1977. *The Process of Education*. Harvard University Press.

[25] Candygram. *http://candygram.sourceforge.net/*. Accessed: 08-02-2010.

[26] Chesebrough, R.A. and Turner, I. 2010. Parallel computing: at the interface of high school and industry. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 280-284.

[27] Clojure - home. *http://clojure.org/*. Accessed: 08-06-2010.

[28] Colombo, C., Del Bimbo, A. et al. 2008. A real-time full body tracking and humanoid animation system. *PARALLEL COMPUTING -AMSTERDAM-*. 34, 12 (2008), 718-726.

[29] Communicating Sequential Processes for Java (JCSP). *http://www.cs.kent.ac.uk/projects/ofa/jcsp/*. Accessed: 08-06-2010.

[30] Computer Science Unplugged. *http://csunplugged.org/*. Accessed: 08-02-2010.

[31] Computing Frontiers 2010 - Workshop. *http://www.computingfrontiers.org/2010/workshop.html*. Accessed: 06-27-2010.

[32] ConcurrentComputing < CourseMaterial < TWiki. *http://wiki.opensparc.net/bin/view.pl/CourseMaterial/ConcurrentComputing*. Accessed: 06-26-2010.

[33] Cooper, S., Dann, W. et al. 2003. Teaching objects-first in introductory computer science. *Proceedings of the 34th Technical Symposium on Computer Science Education (SIGCSE'03)* (2003), 191–195.

[34] CUDA for Finance. *http://www.nvidia.com/object/cuda_finance.html*. Accessed: 07-13-2010.

[35] CUDA for Medical. *http://www.nvidia.com/object/cuda_medical.html*. Accessed: 07-13-2010.

[36] CUDA Zone. *http://www.nvidia.com/object/cuda_home_new.html*. Accessed: 06-30-2010.

[37] Darema, F. 2001. The SPMD Model : Past, Present and Future. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg. 1.

[38] Darling, A., Carey, L. et al. 2003. The Design, Implementation, and Evaluation of mpiBLAST. *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution 2003* (San Jose, California, June 2003).

[39] Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. *OSDI* (2004), 137-150.

[40] Deuce STM - Java Software Transactional Memory. *http://www.deucestm.org/*. Accessed: 08-02-2010.

[41] DiBiasio, D., Clark, W. et al. 1999. Evaluation of a spiral curriculum for engineering. *Frontiers in Education, Annual* (Los Alamitos, CA, USA, 1999), 12D1/15-12D1/18vol.2.

[42] Dongarra, J., Foster, I. et al. 2002. *The Sourcebook of Parallel Computing*. Morgan Kaufmann.

[43] Enable Safe, Scalable Parallelism with Intel Threading Building Block's Concurrent Containers. *http://www.devx.com/cplus/Article/33334*. Accessed: 08-04-2010.

[44] Erlang Programming Language, Official Website. *http://www.erlang.org/index.html*. Accessed: 08-02-2010.

[45] Ernst, D.J. and Stevenson, D.E. 2008. Concurrent CS: preparing students for a multicore world. *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (Madrid, Spain, 2008), 230-234.

[46] Explicit Multi-Threading (XMT) - Home Page. *http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml*. Accessed: 06-28-2010.

[47] Feature - John Shalf talks parallel programming languages. *http://www.isgtw.org/?pid=1002557*. Accessed: 07-27-2010.

[48] Finkel, R.A. and Fishburn, J.P. 1982. Parallelism in alpha-beta search. *Artificial Intelligence*. 19, 1 (Sep. 1982), 89-106.

[49] Flynn, M.J. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*. 100, (1972), 21.

[50] Folding@home. *http://folding.stanford.edu/*. Accessed: 08-06-2010.

[51] Fraser, K. and Harris, T. 2007. Concurrent programming without locks. *ACM Trans. Comput. Syst.* 25, 2 (2007), 5.

[52] Friedman, D.P. and Wand, M. 2008. *Essentials of Programming Languages, 3rd Edition*. The MIT Press.

[53] Gamma, E., Helm, R. et al. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA.

[54] Garrity, P. and Yates, T. *WebMapReduce*.

[55] Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (1985), 80-112.

[56] GlobalInterpreterLock - PythonInfo Wiki. *http://wiki.python.org/moin/GlobalInterpreterLock*. Accessed: 08-06-2010.

[57] Goetz, B., Peierls, T. et al. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional.

[58] Halloway, S. 2009. *Programming Clojure*. Pragmatic Bookshelf.

[59] Haskell - HaskellWiki. *http://www.haskell.org/*. Accessed: 08-02-2010.

[60] Hewitt, C., Bishop, P. et al. 1973. A universal modular ACTOR formalism for artificial intelligence. *Proceedings of the 3rd international joint conference on Artificial intelligence* (Stanford, USA, 1973), 235-245.

[61] Hoare, C. 1985. *Communicating sequential processes*. Prentice/Hall International.

[62] HPCwire: Intel Unveils 48-Core Research Chip. *http://www.hpcwire.com/features/Intel-Unveils-48-Core-Research-Chip-78378487.html*. Accessed: 06-27-2010.

[63] Huang, K. and Thulasiram, R. 2005. Parallel algorithm for pricing American Asian options with multi-

dimensional assets. *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on* (2005), 177-185.

[64] InfoQ: A Pattern Language for Parallel Programming. *http://www.infoq.com/presentations/Pattern-Language-Parallel-Programming*. Accessed: 06-28-2010.

[65] Intel Threaded Building Blocks. *http://www.threadingbuildingblocks.org/*. Accessed: 08-05-2010.

[66] Intel® C++ STM Compiler, Prototype Edition 3.0 - Intel® Software Network. *http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/*. Accessed: 07-16-2010.

[67] Intel® Manycore Testing Lab - Intel® Software Network. *http://software.intel.com/en-us/articles/intel-many-core-testing-lab/*. Accessed: 07-28-2010.

[68] Interim Review Task Force. CS2008 Curriculum Update-http://www.acm.org/education/curricula/ComputerScience2008.pdf.

[69] IronPython.net. *http://ironpython.net/*. Accessed: 08-03-2010.

[70] Joiner, D.A., Gray, P. et al. 2006. Teaching parallel computing to science faculty: best practices and common pitfalls. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, New York, USA, 2006), 239-246.

[71] Joint Task Force on Computing Curricula 2001. Computing curricula 2001. *J. Educ. Resour. Comput.* 1, 3es (2001), 1.

[72] Jones, S.P. 2007. Beautiful Concurrency. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media. 385-406.

[73] Keller, J., Keller, C. et al. 2000. *Practical PRAM Programming*. Wiley-Interscience.

[74] Keutzer, K. and Mattson, T. 2009. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. *ParaPLoP Workshop on Parallel Programming Patterns* (2009).

[75] Klette, R., Huang, T. et al. 2001. *Multi-Image Analysis: 10th International Workshop on Theoretical Foundations of Computer Vision Dagstuhl Castle, Germany, March 12-17, 2000 Revised Papers*. Springer.

[76] Knight, T. 1986. An architecture for mostly functional languages. *Proceedings of the 1986 ACM conference on LISP and functional programming* (Cambridge, Massachusetts, United States, 1986), 105-112.

[77] Kolb, A. and Kolb, D. 2009. The Learning Way. *Simulation & Gaming*. 40, 3 (Jun. 2009), 297-327.

[78] Krasner, G.E. and Pope, S.T. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (1988), 26-49.

[79] Kumar, V., Grama, A. et al. 1994. *Introduction to parallel computing: design and analysis of algorithms*. The Benjamin/Cummings.

[80] Lewandowski, G., Bouvier, D.J. et al. 2010. Commonsense understanding of concurrency: computing students and concert tickets. *Commun. ACM.* 53, 7 (2010), 60-70.

[81] Manger, R., Grbic, M. et al. 1995. A parallel SVD algorithm and its application to financial ratio analysis. *Microprocessing and Microprogramming*. 41, 1 (Apr. 1995), 97-106.

[82] Mattson, T.G., Sanders, B.A. et al. 2004. *Patterns for Parallel Programming*. Addison-Wesley Professional.

[83] Mattson, T.G., Wijngaart, R.V.D. et al. 2008. Programming the Intel 80-core network-on-a-chip terascale processor. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Austin, Texas, 2008).

[84] McConnell, J.J. 2005. Active and cooperative learning: more tips and tricks (part II). *SIGCSE Bull.* 37, 4 (2005), 34–38.

[85] McConnell, J.J. 2005. Active and cooperative learning: tips and tricks (part I). *SIGCSE Bull.* 37, 2 (2005), 27–30.

[86] McConnell, J.J. 2006. Active and cooperative learning: final tips and tricks (part IV). *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education* (New York, NY, USA, 2006), 25–28.

[87] McConnell, J.J. 2006. Active and cooperative learning: further tips and tricks (part 3). *SIGCSE Bull.* 38, 2 (2006), 24–28.

[88] McConnell, J.J. 1996. Active learning and its use in computer science. *Proceedings of the 1st conference on Integrating technology into computer science education* (Barcelona, Spain, 1996), 52-54.

[89] Merigot, A. and Petrosino, A. 2008. Parallel processing for image and video processing: Issues and challenges. *Parallel Comput.* 34, 12 (2008), 694-699.

[90] Miller, R. and Boxer, L. 2005. *Algorithms Sequential & Parallel: A Unified Approach*. Charles River Media.

[91] Milner, R. 1999. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press.

[92] Milner, R., Parrow, J. et al. 1992. A calculus of mobile processes, I. *Inf. Comput.* 100, 1 (1992), 1-40.

[93] Milner, R., Parrow, J. et al. 1992. A calculus of mobile processes, II. *Inf. Comput.* 100, 1 (1992), 41-77.

[94] Mitchell, S.M. and Lutters, W.G. 2006. Assessing the Value of Computer Science Course Material Repositories. *Conference on Software Engineering Education and Training Workshops* (Los Alamitos, CA, USA, 2006), 2.

[95] Neeman, H., Severini, H. et al. 2010. Teaching high performance computing via videoconferencing. *ACM Inroads*. 1, 1 (2010), 67-71.

[96] Odersky, M., Spoon, L. et al. 2008. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc.

[97] Parallel Computing in the Computer Science Curriculum. *http://csinparallel.org*. Accessed: 08-03-2010.

[98] Parallel Java Library. *http://www.cs.rit.edu/~ark/pj.shtml*. Accessed: 08-06-2010.

[99] Pettersson, E., Lundeberg, J. et al. 2009. Generations of sequencing technologies. *Genomics*. 93, 2 (Feb. 2009), 105-111.

[100] Pinson, L. and Wiener, R. 1988. *An introduction to object-oriented programming and Smalltalk*. Addison-Wesley Pub. Co.

[101] Quinn, M.J. 2003. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill Higher Education.

[102] Remagnino, P., Jones, G.A. et al. 2001. *Video-Based Surveillance Systems: Computer Vision and Distributed Processing*. Springer.

[103] Resnick, M., Maloney, J. et al. 2009. Scratch: programming for all. *Commun. ACM*. 52, 11 (2009), 60-67.

[104] Rivoire, S. 2010. A breadth-first course in multicore and manycore programming. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 214-218.

[105] Schwill, A. 1997. Computer science education based on fundamental ideas. *Information technology: supporting change through teacher education*. Chapman & Hall. 285-291.

[106] SETI@home. *http://setiathome.berkeley.edu/*. Accessed: 08-06-2010.

[107] Silberschatz, A., Galvin, P.B. et al. 2008. *Operating System Concepts*. Wiley.

[108] Silven, O. and Jyrkkä, K. 2007. Observations on power-efficiency trends in mobile communication devices. *EURASIP J. Embedded Syst.* 2007, 1 (2007), 17-17.

[109] Single-chip Cloud Computer. *http://techresearch.intel.com/articles/Tera-Scale/1826.htm*. Accessed: 08-02-2010.

[110] Sodan, A.C., Machina, J. et al. 2010. Parallelism via Multithreaded and Multicore CPUs. *Computer*. 43, 3 (2010), 24-32.

[111] Sosic, R. and Gu, J. 1994. Efficient Local Search with Conflict Minimization: A Case Study of the n-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*. 6, 5 (Oct. 1994).

[112] Srinivasan, S. and Mycroft, A. 2008. Kilim: Isolation-typed actors for java. *ECOOP 2008–Object-Oriented Programming*. (2008), 104–128.

[113] Stechert, P. and Schubert, S. 2007. A strategy to structure the learning process towards understanding of informatics systems. *Working/Joint IFIP-Conference Informatics, Mathematics and ICT (IMICT2007): A golden triangle. Boston, USA, 27th–29th June* (2007).

[114] Teach Parallel on Intel Software Network TV - Intel® Software Network. *http://software.intel.com/en-us/articles/teach-parallel/*. Accessed: 08-07-2010.

[115] The Jython Project. *http://jython.org/*. Accessed: 08-03-2010.

[116] The Scala Programming Language. *http://www.scala-lang.org/*. Accessed: 08-06-2010.

[117] The Trilinos Project. *http://trilinos.sandia.gov/*. Accessed: 08-04-2010.

[118] Torbert, S., Vishkin, U. et al. 2010. Is teaching parallel algorithmic thinking to high school students possible?: one teacher's experience. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 290-294.

[119] Van Berkel, C.H. 2009. Multi-core for mobile phones. *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)* (2009).

[120] Van Roy, P. and Haridi, S. 2004. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press.

[121] Vangal, S., Howard, J. et al. 2008. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*. 43, 1 (2008), 29-41.

[122] Venter, J.C., Remington, K. et al. 2004. Environmental genome shotgun sequencing of the Sargasso Sea. *Science (New York, N.Y.)*. 304, 5667 (Apr. 2004), 66-74.

[123] Vishkin, U. 1985. Optimal parallel pattern matching in strings. *Information and Control*. 67, 1-3 (1985), 91-113.

[124] Wheeler, D.A., Srinivasan, M. et al. 2008. The complete genome of an individual by massively parallel DNA sequencing. *Nature*. 452, 7189 (Apr. 2008), 872-876.

[125] Wrinn, M. 2008. Confronting manycore: Parallel programming beyond SMP (multicore is just the beginning); Vendor session. *The 39th ACM technical symposium on Computer science education* (2008).

[126] Wrinn, M. 2010. Suddenly, all computing is parallel: seizing opportunity amid the clamor. *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010), 560-560.

[127] Xavier, C. and Iyengar, S. 1998. *Introduction to parallel algorithms*. Wiley.