NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates

Version I[1]
Dec 2012
Website: http://www.cs.gsu.edu/~tcpp/curriculum/index.php

Curriculum Working Group:

Prasad, Sushil K. (Coordinator; Georgia State University),
Chtchelkanova, Almadena (NSF),
Dehne, Frank (Carleton University, Canada),
Gouda, Mohamed (University of Texas, Austin, NSF),
Gupta, Anshul (IBM T.J. Watson Research Center),
Jaja, Joseph (University of Maryland),
Kant, Krishna (NSF, Intel),
La Salle, Anita (NSF),
LeBlanc, Richard (Seattle University),
Lumsdaine, Andrew (Indiana University),
Padua, David (University of Illinois at Urbana-Champaign),
Parashar, Manish (Rutgers),
Prasanna, Viktor (University of Southern California),
Robert, Yves (INRIA, France),
Rosenberg, Arnold (Northeastern University),
Sahni, Sartaj (University of Florida),
Shirazi, Behrooz (Washington State University),
Sussman, Alan (University of Maryland),
Weems, Chip (University of Massachusetts), and
Wu, Jie (Temple University)

[1] A preliminary version was released in Dec 2010. Contact: Sushil K.Prasad, sprasad@gsu.edu

Abstract:

This report is a revised version of our preliminary report that was released in Dec 2010. The report contains the core topics in parallel and distributed computing (PDC) that a student graduating with a Bachelor's degree in Computer Science or Computer Engineering is expected to have covered. The topics are organized into four areas of architecture, programming, algorithms, and cross-cutting and advanced topics. Additional elective topics are expected. This report is expected to engage the various stakeholders for their adoption and others usage as well as their feedback to periodically update the proposed curriculum. This document contains an introductory write up on curriculum's need and rationale, followed by the proposed topics, with specified Bloom level of coverage, learning outcomes, and suggested number of hours and relevant courses, and additional material in the appendices on suggestions on how to teach individual topics, a cross-reference matrix on core courses vs. topics, and a sample elective course. The revision has updated all the sections, with a new section on rationale for cross cutting topics, reorganization of the programming topics, and updates to several learning outcomes, expected number of hours, and the appendix on how to teach. The scribes of the revised version were Anshul Gupta, Krishna Kant (co-coordinator: crosscutting topics), Andrew Lumsdaine (co-coordinator: crosscutting topics), David Padua (co-coordinator: programming), Sushil Prasad, Yves Robert, Arnold Rosenberg (coordinator: algorithms), Alan Sussman (co-coordinator: programming) and Chip Weems (coordinator: architecture).

Since the release of the preliminary version, we have selected about 80 early adopters from U.S. and from across the world in order to get its evaluation and to obtain templates on how these topics can be adopted in various courses across the curriculum. The early adopters have been awarded stipends, equipment, and travel support through four rounds of competitions (Spring and Fall of 2011 and 2012) with support from NSF, Intel, and nVIDIA. Additional competitions are planned for Fall 2013 and Fall 2014 with proposal deadlines in preceding June. These early adopters and the community meet annually at EduPar workshop held at IPDPS conference to provide feedback and discuss state of art in PDC education. In Fall 2012, NSF supported creating a Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER) for long-term sustenance of this initiative. CEDR center will coordinate the ongoing activities including periodic curricular updates as well as become a hub for collection and creation of educational resources and will facilitate access to hardware and software resources for instruction.

Table of Contents

1. Introduction

Parallel and Distributed Computing (PDC) now permeates most computing activities - the "explicit" ones, in which a person works explicitly on programming a computing device, and the "implicit" ones, in which a person uses everyday tools such as word processors and browsers that incorporate PDC below the user's visibility threshold. The penetration of PDC into the daily lives of both "explicit" and "implicit" users has made it imperative that users be able to depend on the effectiveness, efficiency, and reliability of this technology. The increasing presence of computing devices that contain multiple cores and general-purpose graphics processing units (GPUs) in PCs, laptops, and now even handhelds has empowered even common users to make valued, innovative contributions to the technology of computing. Certainly, it is no longer sufficient for even basic programmers to acquire only the traditional, conventional sequential programming skills. The preceding trends point to the need for imparting a broad-based skill set in PDC technology at various levels in the educational fabric woven by Computer Science (CS) and Computer Engineering (CE) programs as well as related computational disciplines. However, the rapid change in computing hardware platforms and devices, languages, supporting programming environments, and research advances, more than ever challenge educators in knowing what to teach in any given semester in a student's program. Students and their employers face similar challenges regarding what constitutes basic expertise.

Our vision for our committee is one of stakeholder experts working together and periodically providing guidance on restructuring standard curricula across various courses and modules related to parallel and distributed computing. A primary benefit would be for CS/CE students and their instructors to receive periodic guidelines that identify aspects of PDC that are important to cover, and suggested specific core courses in which their coverage might find an appropriate context. New programs at colleges (nationally and internationally) will receive guidance in setting up courses and/or integrating parallelism within the Computer Science, Computer Engineering, or Computational Science curriculum. Employers would have a better sense of what they can expect from students in the area of parallel and distributed computing skills. Curriculum guidelines will similarly help inform retraining and certification for existing professionals.

As background preparation for the development of this curriculum proposal, a planning workshop funded by the National Science Foundation (NSF) was held in February, 2010, in Washington, DC; this was followed up by a second workshop in Atlanta, alongside the IPDPS (International Parallel and Distributed Processing Symposium) conference in April, 2010. These meetings were devoted to exploring the state of existing curricula relating to PDC, assessing needs, and recommending an action plan and mechanisms for addressing the curricular needs in the short and long terms. The planning workshops and their related activities benefited from experts from various stakeholders, including instructors, authors, industry, professional societies, NSF, and the ACM education council. The primary task identified was to propose a set of core topics in parallel and distributed computing for undergraduate curricula for CS and CE students. Further, it was recognized that, in order to make a timely impact, a sustained effort was warranted. Therefore, a series of weekly/biweekly tele-meetings was begun in May, 2010; the series continued through December, 2010.

The goal of the series of meetings was to propose a PDC core curriculum for CS/CE undergraduates, with the premise that *every* CS/CE undergraduate should achieve a specified skill level regarding PDC-related topics as a result of *required* coursework. One impact of a goal of *universal* competence is that many topics that experts in PDC might consider essential are actually too advanced for inclusion. Early on, our working group's participants realized that the set of PDC-related topics that can be designated *core* in the CS/CE curriculum across a broad range of CS/CE departments is actually quite small, and that any recommendations for inclusion of required topics on PDC would have to be limited to the first two years of coursework. Beyond that point, CS/CE departments generally have diverse requirements and electives, making it quite difficult to mandate universal coverage in any specific area. Recognizing this, we have gone beyond the core curriculum, identifying a number of topics that could be included in advanced and/or elective curricular offerings.

In addition, we recognized that whenever it is proposed that new topics be included in the curriculum, many people automatically assume that something else will need to be taken out. However, for many of the topics we propose, this is not the case. Rather, it is more a matter of changing the approach of teaching traditional topics to encompass the opportunities for "thinking in parallel." For example, when teaching array-search algorithms, it is quite easy to point to places where independent operations could take place in parallel, so that the student's concept of search is opened to that possibility. In a few cases, we are indeed proposing material that will require making choices about what it will replace in existing courses. But because we only suggest *potential* places in a curriculum where topics can be added, we leave it to individual departments and instructors to decide whether and how coverage of parallelism may displace something else. The resulting reevaluation is an opportunity to review traditional topics, and perhaps shift them to a place of historical significance or promote them to more advanced courses.

A preliminary version of the proposed core curriculum was released in December 2010. We sought *early adopters* of the curriculum for spring and fall terms of 2011 and 2012 in order to get a preliminary evaluation of our proposal. These adopters included: (i) instructors of introductory courses in Parallel and Distributed Computing, (ii) instructors, department chairs, and members of department curriculum committees, who are responsible for core CS/CE courses, and (iii) instructors of general CS/CE core curriculum courses. The proposing instructors are employing and evaluating the proposed curriculum in their courses. 16 institutions were selected and awarded stipend with NSF and Intel support during Spring'11. We organized a follow-up curriculum and education workshop (EduPar-11 at IPDPS, May 16-20, Anchorage) to bring together early adopters and other experts, and collect feedback from the early adopters and the community. For Fall'11, Spring'12 and Fall'12 rounds of competitions, 18, 21, and 24 early adopters, respectively, were selected. EduPar-12 workshop was held as a regular IPDPS'12 satellite workshop, in Shanghai in May 2012, with expanded scope, and EduPar-13 is being organized at IPDPS-13 in Boston.

This document is a revised version of the preliminary report based on interactions with the early adopters and varied stakeholders at EduPar-11 workshop, bi-weekly tele-meetings from August of 2011 through April of 2012, interactions at EduPar-12, and the follow-up CEDR meetings during Fall 2012. In the three main PDC sub-areas of Architecture, Programming, and Algorithms, plus a fourth sub-area composed of Cross-cutting or Advanced Issues, the working group has deliberated upon various topics and subtopics and

their level of coverage, has identified where in current core courses these could be introduced (Appendix I), and has provided examples of how they might be taught (Appendix II). For each topic/subtopic, the process involved the following.

1. Assign a learning level using Bloom's classification[2] using the following notation.[3]
   K= Know the term (basic literacy)
   C = Comprehend so as to paraphrase/illustrate
   A = Apply it in some way (requires operational command)
2. Write learning outcomes.
3. Identify core CS/CE courses where the topic could be covered.
4. Create an illustrative teaching example.
5. Estimate the number of hours needed for coverage based on the illustrative example.

Our larger vision in proposing this curriculum is to enable students to be fully prepared for their future careers in light of the technological shifts and mass marketing of parallelism through multicores, GPUs, and corresponding software environments, and to make a real impact with respect to all of the stakeholders for PDC, including employers, authors, and educators. This curricular guidance and its trajectory, along with periodic feedback and other evaluation data on its adoption and use, will also help to steer companies hiring students and interns, hardware and software vendors, and, of course, authors, instructors, and researchers.

The time is ripe for parallel and distributed computing curriculum standards, but we also recognize that any revision of a core curriculum is a long-term community effort. The CS2013 ACM/IEEE Computer Science Curriculum Joint Task Force has recognized PDC (along with security) as a main thrust area. We are closely interacting with the Task Force, providing expert feedback on the PDC portion of their initial draft on PDC in Oct, 2011. We will continue to engage with this and other education-oriented task forces in the hope of having significant impact on the CS/CE academic community. More details and workshop proceedings are available at the Curriculum Initiative's website: **http://www.cs.gsu.edu/~tcpp/curriculum/index.php** (email contact: sprasad@gsu.edu).

The rest of this document is organized as follows. First, we provide a general rationale for developing a PDC curriculum (Section 2). We then address the question of whether there is a core set of topics that every student should know. The initial overview concludes with an explanation of how to read the curriculum proposal in a manner consistent with its underlying intent (Section 3). Sections 4,

[2] (i) Anderson, L.W., & Krathwohl (Eds.). (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman, (ii) Huitt, W. (2009). Bloom et al.'s taxonomy of the cognitive domain. *Educational Psychology Interactive*. Valdosta, GA: Valdosta State University. http://www.edpsycinteractive.org/topics/cogsys/bloom.html.

[3] Some advanced topics are identified as "N" as being "not in core" but which may included in an elective course.

5, 6, and 7 then continues with a rationale for each of the four major topic areas in the proposal: architecture, programming, algorithms, and cross-cutting. The proposed curriculum appears in Section 8. For the benefit of potential instructors, Appendix I contains a cross-reference matrix indicating topics for each core course. Appendix II contains suggestions for how to teach individual topics. Finally, Appendix III contains a sample syllabus for an introductory course on parallel and distributed computing. Additional sample courses will be collected at the curriculum website.

2. Why a Parallel and Distributed Computing Curriculum?

2.1 A short history of computing

In the beginning, there was the von Neumann architecture. The first digital computers comprised a functional unit (the processor) that communicated with a memory unit. Humans interfaced with a computer using highly artificial "languages." Humans seldom interfaced with more than one computer at a time, and computers never interfaced with one another. Much of the evolution of digital computers in the roughly seven decades of their existence has brought technical improvements: functional units and memories have become dramatically faster; languages have become dramatically more congenial to the human user. In the earliest days, it seemed as though one could speed up digital computers almost without limit by improving technology: in rapid succession, vacuum tubes gave way to solid-state devices, and these were in turn replaced by integrated circuits --- notably, by using *Very Large Scale Integrated Circuit Technology* (VLSI). One could speed up VLSI circuits impressively by shrinking "feature sizes" within the circuits and by increasing clock rates. Despite these impressive improvements, one could see hints of "handwriting on the wall": The fastest integrated circuits were "hot," presaging that power-related issues (e.g., heat dissipation) would become significant before too long; shrinking feature sizes would ultimately run up against the immutable sizes of atoms. As early as the 1960s, visionaries working along one branch of digital computers' evolutionary tree began envisioning an alternative road toward faster digital computers --- the replication of computer components and the development of tools that allowed multiple components to cooperate in the activity of what came to be called *digital computing*.

The first digital computers that deviated from the von Neumann architecture can be viewed as *hydras* (in analogy with the mythical beast): they were essentially von-Neumann-esque computers that had multiple processors. This development enabled faster computing - several instructions could be executed simultaneously - but they also forced the human user (by now called the *programmer*) to pay attention to coordination among the processors.

An important offspring of the hydra-like *shared-memory computers* had multiple memory boxes in addition to multiple processors. For efficiency, certain processors had preferential (in terms of speed) access to certain memory boxes - which introduced *locality* to the growing list of concerns the programmer had to deal with. Additionally, since each processor-memory box pair could function as an independent von Neumann computer, the programmer now had to orchestrate *communication* among the computers - which "talked" to one another across an *interconnection network*.

It was a short conceptual leap from the preceding "multiple computers in a box" computing platform to *clusters* whose computers resided "close" to one another and intercommunicated over a *local area network* (*LAN*). Among the added concerns arising from the evolution of clusters was the need to account for the greater variability in the *latency* of inter-computer communications. So-called "parallel" computing was beginning to sport many of the characteristics of *distributed* computing, wherein computers share no physical proximity at all.

Perhaps the ultimate step in this evolution has been the development, under a variety of names, of *Internet-based collaborative computing*, wherein geographically dispersed (multi-)computers intercommunicate over the Internet in order to cooperatively solve individual computing problems. Issues such as trust and temporal predictability now join the panoply of other concerns that a programmer must deal with.

Into all of these advances, architects have mixed detailed technical concepts such as multithreading, pipelining, superscalar issue, and short-vector instructions. All of this *heterogeneous* parallelism is now wrapped into commonly encountered computing platforms - in addition to the growing use of vector-threaded co-processors for graphics and scientific computing.

Programming languages have tended to follow an evolutionary path not unlike that of hardware. There have been many attempts to create languages that support abstract models of parallelism, or that correlate with specific parallel architectures, but most have met with only limited success. Even so, popular languages have gradually moved to incorporate parallelism, and languages that focus on various modalities of parallelism have gained a modicum of popularity, so that today it is difficult to ignore parallel computing in even the core of a CS or CE undergraduate programming curriculum. Indeed, we propose that it is a disservice to students not to build a substantial dose of parallel computing into this core.

In the past, it was possible to relegate issues regarding parallelism - such as coordination and locality - to advanced courses that treat subjects such as operating systems, databases, and high performance computing: the issues could safely be ignored in the first years of a computing curriculum. But current-day changes in architecture are driving advances in languages that necessitate new problem solving skills and knowledge of parallel and distributed processing algorithms at even the earliest stages of an undergraduate career. This work is our response to these changes.

2.2 What should every (computer science/engineering) student know about computing?

It has been decades since it was "easy" to supply undergraduates with everything that they need to know about computing as they venture forth into the workforce. This challenge has become ever more daunting with each successive stage of the evolution described

in the preceding section. In addition to enabling undergraduates to understand the fundamentals of "von Neumann computing," we must now prepare them for the very dynamic world of parallel and distributed computing.

This curriculum proposal seeks to address this challenge in a manner that is flexible and broad, always allowing for local variations in emphasis. The field of PDC is changing too rapidly for any proposal with any rigidity to remain valuable to the community for a useful length of time. But it is essential that curricula begin the process of incorporating parallel thinking into the core courses. Thus, the proposal attempts to identify basic concepts and learning goals that are likely to retain their relevance for the foreseeable future.

We see PDC topics as being most appropriately sprinkled throughout a CS/CE curriculum in a way that enhances what is already taught and that melds parallel and distributed computing with existing material in whatever ways are most natural for a given institution/program. While advocating the thesis that relegating PDC subjects to a separate course is not the best means to shift the mindset of students away from purely sequential thinking, we recognize that the separate-course route may work better for some programs.

3. How to Read this Proposal

The reader of this proposal should keep in mind that many of the topics we discuss can appear at multiple levels in the broader curriculum. Upon seeing a topic, the reader should not make a premature judgment regarding its suitability for an undergraduate course. Rather, the reader should consider where and how aspects of the topic might naturally be blended into a suitable context to facilitate the move to holistically allowing students to develop a capacity for parallel and distributed thinking.

For each of the topics, we suggest where and how it can be covered in a curriculum. We thus provide suggestions and examples rather than prescriptions: We are not saying that this is the preferred form of coverage. Our goal is to illustrate one possibility and to get the reader thinking about alternate possibilities.

Our curricular guidelines are not meant to specify precisely where each item is addressed. Our intention is, rather, to encourage instructors to find as many ways as appropriate to insert coverage of the indicated PDC topics into core courses. Even side comments of a few sentences about how some topic under discussion takes on a new perspective in a parallel or distributed context when judiciously sprinkled throughout a course will help students to expand their thinking. Students will more naturally start to think in parallel and distributed terms when they sense that their instructors are always conscious of the implications of parallelism and distributed computing with respect to each topic that is covered in their courses - even topics that have no obvious parallel or distributed content.

This curriculum guide should thus be taken as a basic list of those topics associated with parallel and distribute computing to be kept in mind in the teaching of computer science and engineering. They form a fundamental set of concepts that all students should be familiar with, through seeing them in multiple contexts and at differing levels of sophistication during their academic careers.

In the next four sections we present rationales for the four areas of computer science and engineering into which we have divided the learning goals of the proposed curriculum: Architecture, Programming, Algorithms, and Crosscutting Topics.

4. Rationale for Architecture Topics

Existing computer science and engineering curricula generally include the topic of computer architecture. Coverage may range from a required course to a distributed set of concepts that are addressed across multiple courses.

As an example of the latter, consider an early programming course where the instructor introduces parameter passing by explaining that computer memory is divided into cells, each with a unique address. The instructor could then go on to show how indirect addressing uses the value stored in one such cell as the address from which to fetch a value for use in a computation. There are many concepts from computer architecture bound up in this explanation of a programming language construct.

While the recommended topics of parallel architecture could be gathered into an upper level course and given explicit treatment, they can be similarly interwoven in lower-level courses. Because multicore, multithreaded designs with vector extensions are now mainstream, more languages and algorithms are moving to support data and thread parallelism. Thus, students are going to naturally start bumping into parallel architecture concepts earlier in their core courses.

Similarly, with their experience of social networking, cloud computing, and ubiquitous access to the Internet, students are familiar users of distributed computation, and so it is natural for them to want to understand how architecture supports these applications. Opportunities arise at many points, even in discussing remote access to departmental servers for homework, to drop in remarks that open student's eyes with respect to hardware support for distributed computing.

Introducing parallel and distributed architecture into the undergraduate curriculum goes hand in hand with adding topics in programming and algorithms. Because practical languages and algorithms bear a relationship to what happens in hardware, explaining the reasoning behind a language construct, or why one algorithmic approach is chosen over another will involve a connection with architecture.

A shift to "thinking in parallel" is often cited as a prerequisite for the transition to widespread use of parallelism. The architecture curriculum described here anticipates that this shift will be holistic in nature, and that many of the fundamental concepts of parallelism and distribution will be interwoven among traditional topics.

There are many architecture topics that could be included, but the goal is to identify those that most directly impact and inform undergraduates, and which are well established and likely to remain significant over time. For example, GPGPUs are a current hot topic, but even if they lose favor, the underlying mechanisms of multithreading and vector parallelism have been with us for over four decades, and will remain significant because they arise from fundamental issues of hardware construction.

The proposal divides architecture topics into four major areas: Classes of architecture, memory hierarchy, floating-point representation, and performance metrics. It is assumed that floating point representation is already covered in the standard curriculum, and so it has been included here merely to underscore that for high performance parallel computing, where issues of precision, error, and round off are amplified by the scale of the problems being solved, it is important for students to appreciate the limitations of the representation.

Architecture Classes topics are meant to encourage coverage of the major ways in which computation can be carried out in parallel by hardware. Understanding the differences is key to appreciating why different algorithmic and programming paradigms are needed to effectively use different parallel architectures. The classes are organized along two dimensions: control vs. data parallelism, and the degree to which memory is partitioned.

Memory Hierarchy is covered in the traditional curriculum, but when parallelism and distribution come into play, the issues of atomicity, consistency, and coherence affect the programming paradigm, where they appear, for example, in the explanation of thread safe libraries.

Performance Metrics present unique challenges in the presence of PDC because of asynchrony that results in unrepeatable behavior. In particular, it is much harder to approach peak performance of PDC systems than for serial architectures.

Many of the architecture learning goals are listed as belonging to an architecture course. The teaching examples, however, describe ways that they can be introduced in lower level courses. Some topics are indicated as belonging to a second course in architecture or other advanced courses. These are not included in the core curriculum. We have merely included them as guidance for topical coverage in electives, should a department offer such courses.

5. Rationale for Programming Topics

The material is organized into three subtopics: Paradigms and notations, correctness, and performance. We discuss these in separate sections below. A prerequisite for coverage of much of this material is some background in conventional programming. Even though we advocate earlier introduction of parallelism in a student's programming experience, basic algorithmic problem solving skills must still be developed, and we recognize that it may be easier to begin with sequential models. Coverage of parallel algorithms prior to this material would allow the focus to be exclusively on the practical aspects of parallel programming, but they can also be covered at the same time as necessary and appropriate.

Paradigms and Notations: There are different approaches to parallel programming. These can be classified in many different ways. Here we have used two different ways of classifying the models. First, we classify the paradigms by the target machine model: *SIMD* (single instruction multiple data) is the paradigm in which the parallelism is confined to operations on (corresponding) elements of arrays. This linguistic paradigm is at the basis of Streaming SIMD Extension (SSE) or Altivec macros, some database operations, some operations in data structure libraries, and the languages constructs used for vector machines. *Shared-memory* is the paradigm of OpenMP and Intel's Thread Building Blocks, among other examples. *Distributed memory* is the paradigm underlying message passing and the MPI standard. A hybrid model is when any of the previous three paradigms co-exist in a single program. The logical target machine does not have to be identical to the physical machine. For example, a program written according to the distributed memory paradigm can be executed on a shared-memory machine and programs written in the shared-memory paradigm can be executed on a distributed memory machine with appropriate software support (e.g., Intel's Cluster OpenMP). A second way to classify programming approaches is according to the mechanisms that control parallelism. These are (mostly) orthogonal to the first classification. For example, programs in the SPMD (single program multiple data) paradigm can follow a distributed-memory, shared-memory and even the SIMD model from the first classification. The same is true of programs following the data parallel model. The task spawning model can work within a distributed or shared-memory paradigm. The parallel loop form seems to be mainly used with the shared-memory paradigm, but High-Performance Fortran merged the loop model with the distributed memory paradigm. The students are expected to be familiar with several notations (not languages since in many cases support comes from libraries such as MPI and BSPlib). Not all notations need to be covered, but at least one per main paradigm should be. An example collection that provides this coverage would be Java threads, SSE macros, OpenMP, and MPI. Parallel functional languages are optional.

Correctness and semantics: This set of topics presents the material needed to understand the behavior of parallel programs other than the fact that there are activities that take place (or could take place) simultaneously. Material covers:

a. Tasking, including ways to create parallel tasks and the relationship between implicit tasking and explicit tasking (e.g., OpenMP vs. POSIX threads).
b. Synchronization, including critical sections and producer consumer relations.

c. Memory models. This is an extensive topic and several programming languages have their own model. Only the basic ideas are expected to be covered.
d. Concurrency defects and tools to detect them (e.g. Intel's thread checker).

Performance: The final group of topics is about performance - how to organize the computation and the data for the different classes of machines. The topics here are self-explanatory.

6. Rationale for Algorithms Topics

Parallel/Distributed Models and Complexity: It is essential to provide students a firm background in the conceptual underpinnings of parallel and distributed computing (PDC). Not only is parallelism becoming pervasive in computing, but also technology is changing in this dynamic field at a rapid pace. Students whose education is tied too closely to existing - hence, obsolescent - technology will be at a disadvantage.

Paradigm shifts in the computing field are not new. To recapitulate just a few instances from the Introduction: (1) The VLSI revolution of the late 1970s and early 1980s allowed the development of computers with unheard-of levels of parallelism. New problems related to interprocessor *communication* arose, exposing an aspect of parallel computing that could safely be ignored when levels of parallelism were very modest. (2) As clusters of modest processors (or, workstations) joined "multiprocessors-in-a-box" as parallel computing platforms in the early 1990s, two sea changes occurred: (a) computing platforms became *heterogeneous* for the first time, and (b) *communication delays* became impossible to hide via clever latency-hiding strategies. (3) As computational grids became "parallel" computing platforms around the turn of the millennium, the distinction between *parallel* and *distributed* computing became fuzzier than ever before.

Fortunately, in spite of the "leaps and bounds" evolution of parallel computing technology, there exists a core of fundamental algorithmic principles. Many of these principles are largely independent of the details of the underlying platform architecture, and they provide the basis for developing applications on current and (foreseeable) future parallel platforms. Students should be taught how to identify and synthesize fundamental ideas and generally applicable algorithmic principles out of the mass of parallel algorithm expertise and practical implementations developed over the last few decades.

What, however, should be taught under the rubric "conceptual underpinnings of parallel and distributed computing?" Our choices reflect a combination of "persistent truths" and "conceptual flexibility." In the former camp, one strives to impart: (1) an understanding of how one reasons rigorously about the expenditure of computational resources; (2) an appreciation of fundamental computational limitations that transcend details of particular models. In the latter camp, one needs to expose the student to a variety of

"situation-specific" models, with the hope of endowing the student with the ability to generate new models in response to new technologies.

Algorithmic Paradigms: This section acknowledges the folk wisdom that contrasts giving a person a fish and teaching a student how to fish. Algorithmic paradigms lie in the latter camp. We have attempted to enumerate a variety of paradigms whose algorithmic utility has been demonstrated over the course of decades. In some sense, one can view these paradigms as algorithmic analogues of high-level programming languages. The paradigms in our list can be viewed as generating control structures that are readily ported onto a broad range of parallel and distributed computing platforms. Included here are: the well-known *divide-and-conquer paradigm* that is as useful in the world of sequential computing as in the world of parallel and distributed computing (where it is the basis for the *expansive-reductive*, or "MapReduce" computations); the *series-parallel paradigm* that one encounters, e.g., in multi-threaded computing environments; the *scan*, or, *parallel-prefix* operator, which simplifies the specification of algorithms for myriad computational problems; and many others.

Algorithmic problems: Two genres of specific algorithmic problems play such important roles in a variety of computational situations that we view them as essential to an education about parallel and distributed computing. (1) Several of our entries are auxiliary computations that are useful in a variety of settings. Collective communication primitives are fundamental in myriad applications to the distribution of data and the collection of results. Certain basic functions perform important control tasks, especially as parallel computing incorporates many of the characteristics of distributed computing. The process of leader election endows processors with computationally meaningful names that are useful in initiating and coordinating activities at possibly remote sites. The essential process of termination detection is easy when parallelism is modest and processors are physically proximate, but it is a significant challenge in more general environments. (2) Several of our entries are independent computations that are usually sub-procedures of a broad range of large-scale computations. Sorting and selection are always near the top of everyone's list of basic computations. Algorithms that operate on graphs and matrices also occur in almost every application area.

7. Rationale for Crosscutting and Advanced Topics

Cross-cutting topics: This section includes many of the essential conceptual underpinnings of parallel and distributed computing. These topics are naturally part of the architecture, programming and algorithmic areas but are often treated only implicitly. Since some of these topics are so important and so widely applicable to parallel and distributed computing, it is recommended that discussion of these topics be called out explicitly as part of a PDC curriculum. It is further recommended that these crosscutting topics be reinforced as appropriate in the architecture, programming, and algorithm areas. Since they are cross-cutting themes, these topics can help to tie together and unify the various PDC subject areas. Specific topics include concurrency, nondeterminism, locality of reference, fault-tolerance, and energy efficiency. In terms of Bloom classification, locality has been classified as "C" due to its centrality, while the others have been suggested at the "K" level.

Concurrency provides a fundamental way for describing and reasoning about interacting program constructs (e.g., processes or threads). That the individual steps of interacting programs can be ordered (interleaved) in multiple ways leads to nondeterminism. Performance is a key issue for parallel and distributed programs. Many issues related to performance hinge on locality of reference. As practical parallel computers continue to grow in scale, fault-tolerance and energy efficiency become limiting concerns.

Advanced topics: This section includes topics of significant current or emerging interest and/or those that are better suited for advanced courses but may be introduced in lower level courses in a limited way. At the same time, applications of many of these topics will be familiar to students in their every day lives and can thus serve as motivation for deeper inquiry. The set of topics in this category will necessarily be continually evolving as topics mature and even newer topics appear on the scene. Topics current at this time include cluster computing, cloud/grid computing, peer-to-peer computing, distributed transactions, distributed security, web search, social networking, collaborative computing, and pervasive/mobile computing.

8. Proposed Curriculum

8.1 Notation

Absolutely every individual CS/CE undergraduate must be at this level as a result of his or her required coursework

K = Know the term
C = Comprehend so as to paraphrase/illustrate
A = Apply it in some way
N = Not in Core, but can be in an elective course

CORE COURSES:

CS1	Introduction to Computer Programming (First Courses)
CS2	Second Programming Course in the Introductory Sequence
Systems	Intro Systems/Architecture Core Course
DS/A	Data Structures and Algorithms

ADVANCED/ELECTIVE COURSES:

Arch 2	Advanced Elective Course on Architecture
Algo 2	Elective/Advanced Algorithm Design and Analysis (CS7)
Lang	Programming Language/Principles (after introductory sequence)
SwEngg	Software Engineering
ParAlgo	Parallel Algorithms
ParProg	Parallel Programming
Compilers	Compiler Design
Networking	Communication Networks
DistSystems	Distributed Systems

Note: The numbers of hours suggested in the following tables must be interpreted carefully. Within all tables *except for Algorithms,* the number suggested for a given topic represents a cumulative total across a number of higher-level topics. For example, students need to achieve "A" level competence in shared memory programming. The number of hours required for this is not simply the number assigned to the topic "shared memory" under "target machine model" but rather the total of all hours allocated for "shared

memory" across all higher-level topics --- in addition to the hours allocated to related topics such as "SPMD," "tasks and threads," and "synchronization." In contrast, the hours allocated to Algorithms topics represent our estimates of the effort required to achieve the desired level of competence solely within the context of Algorithms instruction. This decision reflects our recognition that many Algorithms topics develop concepts and tools that will pervade the coverage of many disparate non-Algorithms topics --- the specific list of topics varying from institution to institution. The cumulative number of hours to master a topic is, therefore, impossible to estimate in isolation.

8.2 Architecture Topics

Table 1: Architecture

Topics	BLOOM #	HOURS	Where Covered	Learning Outcome
Classes				
Taxonomy	C	0.5	Systems	Flynn's taxonomy, data vs. control parallelism, shared/distributed memory
Data vs. control parallelism				
Superscalar (ILP)	K	0.25 to 1, based on level	Systems	Describe opportunities for multiple instruction issue and execution (different instructions on different data)
SIMD/Vector (e.g., SSE, Cray)	K	0.1 to 0.5	Systems	Describe uses of SIMD/Vector (same operation on multiple data items), e.g., accelerating graphics for games.
Pipelines				
• Single vs. multicycle	K	1 to 2	Systems	Describe basic pipelining process (multiple instructions can execute at the same time), describe stages of instruction execution
• Data and control hazards	N		Compilers (A), Arch 2 (C)	Understand how one pipe stage can depend on a result from another, or delayed branch resolution can start the wrong instructions in a

				pipe, requiring forwarding, stalling, or restarting
• OoO execution	N		Arch 2 (K)	Understand how independent instructions can be rescheduled for better pipeline utilization, and that various tables are needed to ensure RAW, WAR, and WAW hazards are avoided.
Streams (e.g., GPU)	K	0.1 to 0.5	Systems	Know that stream-based architecture exists in GPUs for graphics
Dataflow	N		Arch 2 (K)	Be aware of this alternative execution paradigm
MIMD	K	0.1 to 0.5	Systems	Identify MIMD instances in practice (multicore, cluster, e.g.), and know the difference between execution of tasks and threads
Simultaneous Multi-Threading	K	0.2 to 0.5	Systems	Distinguish SMT from multicore (based on which resources are shared)
Highly Multithreaded (e.g., MTA)	N		Arch 2 (K)	Have an awareness of the potential and limitations of thread level parallelism in different kinds of applications
Multicore	C	0.5 to 1	Systems	Describe how cores share resources (cache, memory) and resolve conflicts
Heterogeneous (e.g., Cell, on-chip GPU)	K	0.1 to 0.5	Systems	Recognize that multicore may not all be the same kind of core.
Shared vs. distributed memory				
SMP	N		Arch 2 (C)	Understand concept of uniform access shared memory architecture
• Buses	C	0.5 to 1	Systems	Single resource, limited bandwidth and latency, snooping, scalability issues
NUMA(Shared Memory)	N			
• CC-NUMA	N		Arch 2 (K)	Be aware that caches in the context of shared memory depend on coherence protocols
• Directory-based CC-NUMA	N		Arch 2 (K)	Be aware that bus-based sharing doesn't scale, and directories offer an alternative
Message passing (no shared memory)	N		Arch 2 (K)	Shared memory architecture breaks down when scaled due to physical limitations (latency, bandwidth) and results in message passing architectures
• Topologies	N		Algo 2 (C)	Various graph topologies - linear, ring, mesh/torus, tree, hypercube, clique, crossbar
• Diameter	N		Algo 2 (C)	Appreciate differences in diameters of various graph topologies
• Latency	K	0.2 to 0.5	Systems	Know the concept, implications for scaling, impact on

				work/communication ratio to achieve speedup
• Bandwidth	K	0.1 to 0.5	Systems	Know the concept, how it limits sharing, and considerations of data movement cost
• Circuit switching	N		Arch 2 or Networking	Know that interprocessor communication can be managed using switches in networks of wires to establish different point-to-point connections, that the topology of the network affects efficiency, and that some connections may block others
• Packet switching	N		Arch 2 or Networking	Know that interprocessor communications can be broken into packets that are redirected at switch nodes in a network, based on header info
• Routing	N		Arch 2 or Networking	Know that messages in a network must follow an algorithm that ensures progress toward their destinations, and be familiar with common techniques such as store-and-forward, or wormhole routing
Memory Hierarchy				
• Cache organization	C	0.2 to 1	Systems	Know the cache hierarchies, shared caches (as opposed to private caches) result in coherency and performance issues for software
• Atomicity	N		Arch 2 (K)	Need for indivisible operations can be covered in programming, OS, or database context
• Consistency	N		Arch 2 (K)	Models for consistent views of data in sharing can be covered in programming, OS, or database context
• Coherence	N		Arch 2 (C)	Describe how cores share cache and resolve conflicts - may be covered in programming. OS, or database context
• False sharing	N		Arch2 (K)/ ParProg (K)	Awareness, examples of how it originates
• Impact on software	N		Arch2 (C)/ ParProg (A)	Issues of cache line length, memory blocks, patterns of array access, compiler optimization levels
Floating point representation				These topics are supposed to be in the ACM/IEEE core curriculum already – they are included here to emphasize their importance, especially in the context of PDC.
Range	K		CS1/CS2/Systems	Understand that range is limited, implications of infinities
Precision	K	0.1 to 0.5	CS1/CS2/Systems	How single and double precision floating point numbers impact software performance
Rounding issues	N		Arch 2 (K)/ Algo 2 (A)	Understand rounding modes, accumulation of error and loss of precision
Error propagation	K	0.1 to 0.5	CS2	Understand NaN, Infinity values and how they affect computations and exception handling

IEEE 754 standard	K	0.5 to 1	CS1/CS2/Systems	Representation, range, precision, rounding, NaN, infinities, subnormals, comparison, effects of casting to other types
Performance metrics				
Cycles per instruction (CPI)	C	0.25 to 1	Systems	Number of clock cycles for instructions, understand the performance of processor implementation, various pipelined implementations
Benchmarks	K	0.25 to 0.5	Systems	Awareness of various benchmarks and how they test different aspects of performance
• Spec mark	K	0.25 to 0.5	Systems	Awareness of pitfalls in relying on averages (different averages can alter perception of which architecture is faster)
• Bandwidth benchmarks	N		Arch 2 (K)	Be aware that there are benchmarks focusing on data movement instead of computation
Peak performance	C	0.1 to 0.5	Systems	Understanding peak performance, how it is rarely valid for estimating real performance, illustrate fallacies
• MIPS/FLOPS	K	0.1	Systems	Understand meaning of terms
Sustained performance	C	0.1 to 0.5	Systems	Know difference between peak and sustained performance, how to define, measure, different benchmarks
• LinPack	N		ParProg (K)	Be aware of the existence of parallel benchmarks

8.3 Programming Topics

Table 2 Programming

Topics	BLOOM#	HOURS	Where Covered	Learning Outcome
Parallel Programming paradigms and Notations				
By the target machine model		*5*		
SIMD	K	0.5	CS2; Systems	Understand common vector operations including element-by-element operations and reductions.
• Processor vector extensions	K		Systems	Know examples - SSE/Altivec macros
• Array language extensions	N		ParProg (A)	Know how to write parallel array code in some language (e.g., Fortran95, Intel's C/C++ Array Extension[CEAN])
Shared memory	A	2.0	CS2; DS/A; Lang	Be able to write correct thread- based programs (protecting shared data) and understand how to obtain speed up.
• Language extensions	K			Know about language extensions for parallel programming. Illustration from Cilk (spawn/join) and Java (Java threads)
• Compiler directives/	C			Understand what simple directives, such as those of OpenMP, mean (parallel for, concurrent section),

pragmas				show examples
• Libraries	C			Know one in detail, and know of the existence of some other example libraries such as Pthreads, Pfunc, Intel's TBB (Thread building blocks), Microsoft's TPL (Task Parallel Library), etc.
Distributed memory	C	1.0	DS/A; Systems	Know basic notions of messaging among processes, different ways of message passing, collective operations
• Message passing	N		ParProg(C)	Know about the overall organization of an message passing program as well as point-to-point and collective communication primitives (e.g., MPI)
• PGAS languages	N		ParProg (C)	Know about partitioned address spaces, other parallel constructs (UPC, CoArray Fortran, X10, Chapel)
Client Server	C	1.0	DS/A; Systems	Know notions of invoking and providing services (e.g., RPC, RMI, web services) - understand these as concurrent processes
Hybrid	K	0.5	Systems	Know the notion of programming over multiple classes of machines simultaneously (CPU, GPU, etc.)
By the control statement				
Task/thread spawning	A	1	CS2; DS/A	Be able to write correct programs with threads, synchronize (fork-join, producer/consumer, etc.), use dynamic threads (in number and possibly recursively) thread creation - (e.g. Pthreads, Java threads, etc.) - builds on shared memory topic above
SPMD	C	1	CS2; DS/A	Understand how SPMD program is written and how it executes
• SPMD notations	C			Know the existence of highly threaded data parallel notations (e.g., CUDA, OpenCL), message passing (e.g, MPI), and some others (e.g., Global Arrays, BSP library)
Data parallel	A	1	CS2; DS/A; Lang	Be able to write a correct data-parallel program for

				shared-memory machines and get speedup, should do an exercise. Understand relation between different notations for data parallel: Array notations, SPMD, and parallel loops. Builds on shared memory topic above.
• Parallel loops for shared memory	A		CS2; DS/A; Lang	Know, through an example, one way to implement parallel loops, understand collision/dependencies across iterations (e.g., OpenMP, Intel's TBB)
• Data parallel for distributed memory	N		ParProg (K)	Know data parallel notations for distributed memory (e.g., High Performance Fortran)
Functional/logic languages	N		ParProg (K)	Understanding advantages and disadvantages of very different programming styles (e.g., Parallel Haskell, Parlog, Erlang)
Semantics and correctness issues				
Tasks and threads	K	0.5	CS2; DS/A; Systems, Lang	Understand what it means to create and assign work to threads/processes in a parallel program, and know of at least one way do that (e.g., OpenMP, Intel TBB, etc.)
Synchronization	A	1.5	CS2; DS/A; Systems	Be able to write shared memory programs with critical regions, producer- consumer communication, and get speedup; know the notions of mechanisms for concurrency (monitors, semaphores, etc. - [from ACM 2008])
• Critical regions	A			Be able to write shared memory programs that use critical regions for synchronization
• Producer-consumer	A			Be able to write shared memory programs that use the producer-consumer pattern to share data and synchronize threads
• Monitors	K			Understand how to use monitors for synchronization
Concurrency defects	C	1.0	DS/A; Systems	Understand the notions of deadlock (detection, prevention), race conditions (definition),

				determinacy/non-determinacy in parallel programs (e.g., if there is a data race, the output may depend on the order of execution)
• Deadlocks	C			Understand what a deadlock is, and methods for detecting and preventing them
• Data Races	K			Know what a data race is, and how to use synchronization to prevent it
Memory models	N		ParProg (C)	Know what a memory model is, and the implications of the difference between strict and relaxed models (performance vs. ease of use)
• Sequential consistency	N			Understand semantics of sequential consistency for shared memory programs
• Relaxed consistency	N			Understand semantics of one relaxed consistency model (e.g., release consistency) for shared memory programs
Tools to detect concurrency defects	K	0.5	DS/A; Systems	Know the existence of tools to detect race conditions (e.g., Eraser)
Performance issues				
Computation	C	1.5	CS2; DS/A	Understand the basic notions of static and dynamic scheduling, mapping and impact of load balancing on performance
Computation decomposition strategies	C			Understand different ways to assign computations to threads or processes
• Owner computes rule	C			Understand how to assign loop iterations to threads based on which thread/process owns the data element(s) written in an iteration
• Decomposition into atomic tasks	C			Understand how to decompose computations into tasks with communication only at the beginning and end of each task, and assign them to threads/processes
• Work stealing	N		ParProg (C)	Understand one way to do dynamic assignment of computations

Program transformations	N		Compilers (A)	Be able to perform simple loop transformations by hand, and understand how that impacts performance of the resulting code (e.g., loop fusion, fission, skewing)
Load balancing	C	1.0	DS/A; Systems	Understand the effects of load imbalances on performance, and ways to balance load across threads or processes
Scheduling and mapping	C	1.0	DS/A; Systems	Understand how a programmer or compiler maps and schedules computations to threads/processes, both statically and dynamically
• Static				Understand how to map and schedule computations before runtime
• Dynamic				Understand how to map and schedule computations at runtime
Data	K	1.0	DS/A; Lang	Understand impact of data distribution, layout and locality on performance; know false sharing and its impact on performance (e.g., in a cyclic mapping in a parallel loop); notion that transfer of data has fixed cost plus bit rate (irrespective of transfer from memory or inter-processor)
Data distribution	K			Know what block, cyclic, and block-cyclic data distributions are, and what it means to distribute data across multiple threads/processes
Data layout	K			Know how to lay out data in memory to get improve performance (memory hierarchy)
Data locality	K			Know what spatial and temporal locality are, and how to organize data to take advantage of them
False sharing	K			Know that for cache coherent shared memory systems, data is kept coherent in blocks, not individual words, and how to avoid false sharing across threads of data for a block
Performance	K	0.5	DS/A; Systems	Know of tools for runtime monitoring (e.g., gprof,

monitoring tools				Vtune)
Performance metrics	C	1.0	CS2; DS/A	Know the basic definitions of performance metrics (speedup, efficiency, work, cost), Amdahl's law; know the notions of scalability
• Speedup	C			Understand how to compute speedup, and what it means
• Efficiency	C			Understand how to compute efficiency, and why it matters
• Amdahl's law	K			Know that speedup is limited by the sequential portion of a parallel program, if problem size is kept fixed
• Gustafson's Law	K			Understand the idea of weak scaling, where problem size increases as the number of processes/threads increases
• Isoefficiency	N		ParProg; Algo2 (C)	Understand the idea of how quickly to increase problem size with number of processes/threads to keep efficiency the same

8.4 Algorithm Topics

Table 3 Algorithms

Note: Recall that the numbers of hours in this table reflect just the coverage within the Algorithms portion of the curriculum. (See the explanatory note in Section 8.1)

Topics	BLOOM #	HOURS	Where Covered	Learning Outcome
Parallel and Distributed Models and Complexity		**7.41**		Be exposed to the models and to the intrinsic degree of parallelism of some elementary key algorithms (e.g., maximum-finding, summation)
Costs of computation:		*1.66*		Follow arguments for parallel time and space complexity given by instructor
Asymptotics	C	1	DS/A	Understand upper (big-O) and lower bounds (big- Omega,); follow elementary big-O analyses, e.g., the O(log n) tree-depth argument for mergesort with unbounded parallelism.
Time	C	0.33	DS/A	Recognize time as a fundamental computational resource that can be influenced by parallelism
Space/Memory	C	0.33	DS/A	Recognize space/memory in the same manner as time
Cost reduction:		*1*		Be exposed to a variety of computational costs other than time that can benefit from parallelism (a more advanced extension of "speedup")
Speedup	C	1	DS/A	Recognize the use of parallelism either to solve a given problem instance faster or to solve larger instance in the same time (strong and weak scaling)
Space compression	N	0.33		Be exposed to ways in which the computational resource "space" behaves the same as "time" and to ways in which the two cost measures differ
Cost tradeoffs:		*0.75*		Recognize the inter-influence of various cost measures
Time vs. space	N	0.5	DS/A	Observe several examples of this prime cost tradeoff; lazy vs. eager evaluation supplies many examples

Power vs. time	N	0.25	DS/A	Observe at least one example of this prime cost tradeoff (the literature on "VLSI computation" --- e.g., the footnoted books[4] [5] --- yield many examples)
Scalability in algorithms and architectures	C/K	0.5	DS/A	Comprehend via several examples that having access more processors does not guarantee faster execution --- the notion of inherent sequentiality (e.g., the seminal paper by Brent)
Model-based notions:		*4*		Recognize that architectural features can influence amenability to parallel cost reduction and the amount of reduction achievable
Notions from complexity-theory:		2		Understand (via examples) that some computational notions transcend the details of any specific model
• PRAM	K	1	DS/A	Recognize the PRAM as embodying the simplest forms of parallel computation: Embarrassingly parallel problems can be sped up easily just by employing many processors.
• BSP/CILK	K	1	DS/A	Be exposed to higher-level algorithmic abstractions that encapsulate more aspects of real architectures. Either BSP or CILK would be a good option to introduce a higher level programming model and higher-level notions. Remark that both of these abstractions have led to programming models.
• Simulation/emulation	N	1	Algo 2	See simple examples of this abstract, formal analogue of the *virtual machines* that are discussed under programming topics. It is important to stress that (different aspects of the same) central notions of PDC can be observed in all four of our main topic areas.
• P-completeness and #P-completeness	N	1	Algo 2	Recognize these two notions as the parallel analogues of NP-completeness. They are the quintessential model-independent complexity-theoretic notions.
• Cellular automata	N	1	Algo 2	Be exposed to this important model that introduces new aspects of parallelism/distributed computing --- possibly via games (such as Life)
Notions from scheduling:		2		Understand how to decompose a problem into tasks
• Dependencies	A	0.5	CS1/CS2, DS/A	Observe how dependencies constrain the execution order of sub-computations --- thereby lifting one from the limited domain of "embarrassing parallelism" to more complex computational structures.

[4] F.T. Leighton (1992): Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, San Mateo, Cal.

[5] J.D. Ullman (1984): Computational Aspects of VLSI. Computer Science Press, Rockville, Md.

• Task graphs	C	0.5	DS/A; SwEngg	See multiple examples of this concrete algorithmic abstraction as a mechanism for exposing inter-task dependencies. These graphs, which are used also in compiler analyses, form the level at which parallelism is exposed and exploited.
• Work	K	0.5	DS/A	Observe the impact of computational work (e.g., the total number of tasks executed) on complexity measures such as power consumption.
• (Make)span	K	0.5	DS/A	Observe analyses in which makespan is identified with parallel time (basically, time to completion)
Algorithmic Paradigms		**4.5**		
Divide & conquer (parallel aspects)	C	*1*	CS2, DS/A, Algo 2	Observe, via tree-structured examples such as mergesort or numerical integration (trapezoid rule, Simpson's rule) or (at a more advanced level) Strassen's matrix-multiply, how the same structure that enables divide and conquer (sequential) algorithms exposes opportunities for parallel computation.
Recursion (parallel aspects)	C	*0.5*	CS2, DS/A	Recognize algorithms that, via unfolding, yield tree structures whose subtrees can be computed independently, in parallel
Scan (parallel-prefix)	N	0.5	ParAlgo, Architecture	Observe, via several examples[6],[7] this "high-level" algorithmic tool
Reduction (map-reduce)	K/C	*1*	DS/A	Recognize, and use, the tree structure implicit in scalar product or mergesort or histogram (equivalent apps)
Stencil-based iteration	N	0.5	ParAlgo	Observe illustrations of mapping and load balancing via stenciling
Dependencies:	K	*0.5*	Systems	Understand the impacts of dependencies
"Oblivious" algorithms	N	0.5	ParAlgo	Observe examples of these "model-independent" algorithms that ignore the details of the platform on which they are executed. Recognize obliviousness as an important avenue toward *portability*.
Blocking	N	0.5	ParAlgo	See examples of this algorithmic manifestation of memory hierarchies
Striping	N	0.5	ParAlgo	See examples of this algorithmic manifestation of memory hierarchies
"Out-of-core" algorithms	N	0.5	ParAlgo	Observe ways of accommodating a memory/storage hierarchy by dealing with issues such as *locality* and acknowledging the changes in cost measures at the various levels of the hierarchy.
Series-parallel composition	C	*1*	CS2(K),	Understand how "barrier synchronizations" can be used to enable a simple

[6] G.E. Blelloch (1989): Scans as primitive parallel operations. IEEE Transactions on Computers 38, pp. 1526—1538

[7] F.T. Leighton (1992): Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, San Mateo, Cal.

			Systems(C)	thread-based abstraction for parallel programming. Understand the possible penalties (in parallelism) that this transformation incurs
Graph embedding as an algorithmic tool	N	1	ParAlgo	Recognize this key algorithmic tool for crafting simulations/emulations.
Algorithmic problems		**8.5**		The important thing here is to emphasize the parallel/distributed aspects of the topic
Communication	C/A	*2*		Understand --- via hands-on experience --- that inter-processor communication is one of the most challenging aspects of PDC.
Broadcast	C/A	1	DS/A	Use this important mode of global communication; observe enabling algorithms for various platforms (e.g., recursive doubling)
Multicast	K/C	0.5	DS/A	Recognize other modalities of global communication on a variety of platforms: e.g., rings, 2D-meshes, hypercubes, trees
Scatter/gather	C/A	0.5	DS/A	Recognize these informational analogues of Map and reduce
Gossip	N	0.5	Dist Systems, Networking	Recognize how all-to-all communication simplifies certain computations
Asynchrony	K	*0.5*	CS2	Understand asynchrony as exhibited on a distributed platform, its strengths (no need for synchs) and pitfalls (the danger of race conditions)
Synchronization	K	*1*	CS2, DS/A	Be aware of methods for controlling race conditions
Sorting	C	*1.5*	CS2, DS/A	Observe several sorting algorithms for varied platforms --- together with analyses. Parallel merge sort is the simplest example, but equally simple alternatives for rings and meshes might be covered also; more sophisticated algorithms might be covered in more advanced courses
Selection	K	*0.5*	CS2, DS/A	Observe algorithms for finding order statistics, notably min and max. Understand that selection can always be accomplished by sorting but that direct algorithms may be simpler.
Graph algorithms:		*1*		
Search	C	1	DS/A	Know how to carry out BFS- and DFS-like parallel search in a graph or solution space
Path selection	N	1		
Specialized computations	A	*2*	CS2, DS/A	Master one or two from among computations such as: matrix product, transposition, convolution, and linear systems; *recognize how algorithm design reflects the structure of the computational problems.*
Convolutions	Optional	1		Be exposed to block or cyclic mappings; understand trade-offs with communication costs
Matrix computations	Optional	1		Understand the mapping and load balancing problems on various platforms

				for significant concrete instances of computational challenges that are discussed at a higher level elsewhere
• Matrix product	Optional	1		Observe a sample "real" parallel algorithm, such as Cannon's algorithm[8]
• Linear systems	Optional	1		Observe load-balancing problems in a concrete setting
• Matrix arithmetic	Optional	1		Observe the challenges in implementing even "simple" arithmetic
• Matrix transpose	Optional	1		Observe a challenging concrete data permutation problem
Termination detection	N/K	1	ParAlgo	See examples that suggest the difficulty of proving that algorithms from various classes actually terminate. For more advanced courses, observe proofs of termination, to understand the conceptual tools needed.
Leader election/symmetry breaking	N/K	2	ParAlgo	Observe simple symmetry-breaking algorithms, say for a PRAM

[8] H. Gupta, P. Sadayappan (1996): Communication Efficient Matrix-Multiplication on Hypercubes. Parallel Computing 22 , pp. 75-99.

8.5 Cross Cutting and Advanced Topics

Table 4 Cross Cutting and Advanced

Topics	BLOOM #	HOURS	Where Covered	Learning Outcome
High level themes:				
Why and what is parallel/distributed computing?	K	0.5	CS1, CS2	Know the common issues and differences between parallel and distributed computing; history and applications. Microscopic level to macroscopic level parallelism in current architectures.
Cross-Cutting topics				know these underlying themes
Locality	C	1	DS/A, Systems	Understand this as a dominant factor impacting performance - minimizing cache/memory access latency or inter-processor communication.
Concurrency	K	0.5	CS2, DS/A	The degree of inherent parallelism in an algorithm, independent of how it is executed on a machine
Non-determinism	K	0.5	DS/A, Systems	Different execution sequences can lead to different results hence algorithm design either be tolerant to such phenomena or be able to take advantage of this.
Power Consumption	K	0.5	Systems, DS/A	Know that power consumption is a metric of growing importance, its impact on architectural evolution, and design of algorithms and software.
Fault tolerance	K	0.5	Systems	Large-scale parallel/distributed hardware/software systems are prone to components failing but system as a whole needs to work.
Performance modeling	N	0.5	Arch 2, Networking, Adv OS	Be able to describe basic performance measures and relationships between them for both individual resources and systems of resources.
Current/Advanced Topics				
Cluster Computing	K	0.25	CS2, DS/A, System	Be able to describe a cluster as a popular local-memory architecture with commodity compute nodes and a high-performance interconnection network.

Cloud/grid Computing	K	0.25	CS2, DS/A, System	Recognize cloud and grid as shared distributed resources - cloud is distinguished by on-demand, virtualized, service-oriented software and hardware resources.
Peer to Peer Computing	K	0.25	CS1, CS2	Be able to describe a peer to peer system and the roles of server and client nodes with distributed data. Recognize existing peer to peer systems.
Consistency in Distributed Transactions	K	0.25	CS1,CS2, Systems	Recognize classic consistency problems. Know that consistency maintenance is a primary issue in transactions issued concurrently by multiple agents.
Web search	K	0.25	CS1, CS2	Recognize popular search engines as large distributed processing systems for information gathering that employ distributed hardware to support efficient response to user searches.
Security in Distributed Systems	K	0.5	Systems	Know that distributed systems are more vulnerable to privacy and security threats; distributed attacks modes; inherent tension between privacy and security.
Social Networking/Context	N	0.5	AI, Distributed Systems, Networking,	Know that the rise of social networking provides new opportunities for enriching distributed computing with human & social context.
Collaborative Computing	N	0.25	HCI, Dist Systems, OS	Know that collaboration between multiple users or devices is a form of distributed computing with application specific requirements.
Performance modeling	N	0.5	Arch 2, Networking	Be able to describe basic performance measures and relationships between them for both individual resources and systems of resources.
Web services	N	0.5	Web Programming, Dist Systems, Adv OS,	Know that web service technology forms the basis of all online user interactions via browser.
Pervasive and Mobile computing	N	0.5	Mobile Computing, Networking, Dist System	Know that the emerging pervasive and mobile computing is another form of distributed computing where context plays a central role.

9. Appendix I: Cross Reference Matrix – Core Courses vs. Topics

For ease of reference by adopters/instructors of core courses, here are the core courses cross-referenced with topics for the four areas, with information extracted from the previous four tables.

Table 5: Algorithm

Topics	DS/A	CS2
• Costs of computation:		
Asymptotics	1	
time	1	
space	1	
speedup	1	
• Cost reduction:		
space compression, etc.		
• Cost tradeoffs:		
time vs. space,	1	
power vs. time, etc.	1	
• Scalability in algorithms and architectures	1	
• Model-based notions:		
– Notions from complexity-theory:		
PRAM	1	
BSP/CILK	1	
simulation/emulation,		
P-completeness,		
#P-completeness		
Cellular automata		
Notions from scheduling		
dependencies,	1	1

task graphs,	1	
work,	1	
(make)span	1	
• Divide & conquer (parallel aspects)	1	1
• Recursion (parallel aspects)	1	1
• Scan (parallel-prefix)		
•reduction (map-reduce)		
• Stencil-based iteration		
• Dependencies:		
"oblivious" algorithms		
blocking		
striping		
"out-of-core" algorithms		
• Series-parallel composition		1
• Graph embedding as an algorithmic tool		
• Communication:		
broadcast,	1	
multicast,	1	
scatter/gather	1	
gossip		
• Asynchrony		1
• Synchronization	1	1
Sorting	1	1
Selection	1	1
• Graph algorithms:		
search		
path selection		
• Specialized computations:	1	1
convolutions		
matrix computations		

matrix product		
linear systems		
matrix arithmetic		
matrix transpose		
Termination detection		
Leader election/symmetry breaking		

Table 6: Programming

Topics	Systems	DS/A	CS2
Parallel Programming paradigms and Notations			
By the target machine model			
SIMD	1		1
Processor vector extensions	1		
Shared memory		1	1
Distributed memory	1	1	
Client Server		1	1
Hybrid	1		
By the control statement			
Task/thread spawning		1	1
SPMD		1	1
Data parallel		1	1
Parallel loops for shared memory		1	1
Semantics and correctness issues			
Tasks and threads		1	1
Synchronization		1	1
Concurrency defects	1	1	
Tools to detect concurrency defects	1	1	
Performance issues			
Computation		1	1

Load balancing	1	1	
Scheduling and mapping	1	1	
Data (distribution, layout, locality)		1	
Performance monitoring tools	1	1	
Performance metrics		1	1

Table 7: Architecture

Topics	CS1	CS2	Systems
Superscalar (ILP)			1
SIMD/Vector (e.g., SSE, Cray)			1
Pipelines			
(Single vs. multicycle)			1
Data and control hazards			
OoO execution			
Streams (e.g., GPU)			1
Dataflow			
MIMD			1
Simultaneous Multithreading (e.g., Hyperthreading)			1
Highly Multithreaded (e.g., MTA)			
Multicore			1
Cluster			
Heterogeneous (e.g., Cell)			1
Grid/cloud		1	
SMP			

Buses			1
NUMA (Shared Memory)			
CC-NUMA			
Directory-based CC-NUMA			
Message passing (no shared memory)			
Topologies			1
Diameter			1
Latency			1
Bandwidth			1
Circuit switching			
Packet switching			
Routing			
Cache organization			1
Atomicity			
Consistency			
Coherence			
False sharing			
Impact on software			
Range	1	1	1
Precision	1	1	1
Rounding issues			
Error propagation			
754 standard			1
cycles per instruction (CPI)			1
Benchmarks			1
Spec mark			1
Bandwidth benchmarks			
Peak performance			1
MIPS/FLOPS			1
Sustained performance			1
LinPack			

Table 8: Crosscutting/Advanced

Topics \ Where Covered	CS1	CS2	Systems	DS/A
High level themes:				
Why and what is parallel/distributed computing?	1	1		
Crosscutting topics:				
Concurrency		1		1
Non-determinism			1	1
Power			1	1
Locality			1	1
Current/Hot/Advanced Topics				
Cluster		1	1	1
cloud/grid		1	1	1
p2p	1	1		
fault tolerance			1	
Security in Distributed System			1	
Distributed transactions	1	1		
web search	1	1		
Social Networking/Context				
Collaborative Computing				
performance modeling				
web services				
pervasive computing				
mobile computing				

10. Appendix II: Suggestions on how to teach topics

10.1 Architecture

Data versus control parallelism:

Superscalar (ILP): Multiple issues can be covered in detail in a compiler class, an intro to systems or assembly language, or in architecture. However, even in an early programming class, students are often curious about the factors that affect performance across different processors, and are receptive to hearing about how different models have different numbers of pipelines and greater or lesser ability to cope with unbalanced and dependent groups of instructions.

SIMD/Vector (e.g., SSE, Cray): This can be mentioned any place that vector/matrix arithmetic algorithms are covered, and even in a data structures class. In an architecture class, it may be part of introducing SSE-style short vector operations in a 64-bit ISA. If chip photos are shown, it can be noted that the control unit section of a processor could be shared among multiple identical ALUs to create an explicitly SIMD architecture. Or, in a survey of supercomputers, Cray vector architectures can be described as a historical example, and the evolution of SIMD to SPMD and streaming data parallel designs can be traced.

Pipelines: Pipelines appear in simple form as one means of implementing a vector multiplier, where stages are essentially identical. Otherwise, they are typically covered in an architecture course. It is possible, however, to introduce the concept earlier in a multithreaded application where a chain of consumer/producer threads feed forward through intermediate buffers or queues.

Single vs. multicycle: The difference between data paths in a single cycle and pipelined processor is most directly treated in an architecture course by walking through the creation of the simpler form and then converting it to a pipeline. However, it can also be shown in an advanced programming course, where the steps of a large, monolithic, task are broken into a chain of threads that each execute in a shorter amount of time, and provide opportunities for the OS to take advantage of multicore capabilities.

Streams (e.g., GPU): Graphics processors are one example of a stream architecture, and can be described in terms of how they marshal a block of identical threads to operate in a sweep over a large array of data, and can be covered in a survey of approaches to data parallelism in an architecture course, or as a performance topic in a graphics course. Streams can also be used as a design pattern for threaded code operating on a large data structure.

MIMD: In a survey of parallelism in an architecture course it is easy to take the step from uniprocessors to multiprocessors, since it is obvious that a CPU can be replicated and used in parallel. In an OS course, multitasking can be described both in a uniprocessor and a

multiprocessor context. In an early programming course, it is likely that students will be curious about what the popular term multicore means, and how it could impact their programming.

Simultaneous Multithreading (e.g., Hyperthreading): In an architecture course, different granularities of multithreading should be addressed. Then the impact on the microarchitecture (multiple register sets, more ALU utilization and increased heat, additional logic to manage exceptions, etc.) can be covered. In an early course, where performance is discussed, it follows naturally after explaining superscalar issue and the low rate at which issue slots are filled. It can then be contrasted with multicore in the sense that it does not replicate the entire CPU, but just the parts that are essential to enabling a second thread to run in a way that fills in some underutilized resources. An analogy is using a truck to tow two trailers versus using two trucks to tow the same two trailers.

Multicore: In an architecture course, the rise in power and heat with increased clock rate will naturally follow the idea that pipelining enables faster clock rates. The limited ability of chips to consume power and dissipate heat then motivates the shift away from the trend of using more real estate to build faster processors toward building more cores that operate at a slower rate. Once a chip has multiple cores, the question of how they work together leads to coverage of communication mechanisms. Scaling up the idea of multicores then leads to the question of whether they must all be identical. Tying chip scaling back to fault and yield models provides an indication that cores are likely to be heterogeneous in performance and functionality as a result of manufacturing variations. At a high level of abstraction, some of these concepts can be explained in an early programming course, where factors that affect performance are discussed.

Cluster: Students may first encounter a cluster in a departmental compute server for use an instructional lab. Or it may be mentioned in Internet course when explaining how services such as search engines and on-line auctions are supported. In an architecture class, they are motivated by the practical limitations of motherboard size, and the ease of assembly using off the shelf components. In this context, students should also be cautioned regarding practical issues of power conditioning, thermal management, and mean time to failure of nodes.

Grid/cloud: Students will typically have personal experience with cloud computing through internet services, such as document storage and sharing, that are available anywhere. Questions about how this works can be addressed in programming (where such services may be used by teams), networking, and database courses. More advanced courses can cover grid issues in terms of latency, availability, load distribution and balancing, allocation policies, etc.

Shared versus distributed memory:

SMP:

Buses: In the earliest courses, students often want to understand the differences in the many kinds of buses they hear about (front side, PCI, USB, etc.), and this presents and opportunity to explain how multiple components in a computer can share a common communication link. In an architecture course, it is normal to encounter various internal buses, including the memory bus. Once the idea of DMA and multiple masters is introduced, it is logically a small step to have two or more processors on the same memory bus. As with I/O devices on the bus, a protocol is needed to ensure that intended recipients receive current data.

Message passing (no shared memory):

Topologies: In an architecture course, once the idea of inter-processor communication over a network link is established, going beyond two processors opens up options for the arrangement of links. A few examples illustrate the potential explosion of topologies, so it is then worth mentioning that most can be simulated with constant order slowdown by a few key topologies. Thus, it boils down to more practical considerations, such as the ease of building a mesh on a circuit board, or wiring network cable and routers in a high-degree fat-tree pattern.

Diameter: Although this can also be covered as part of graph theory, it is useful to show the differences in diameter of a few topologies, mainly so students can see that there are some very poor choices possible, such as linear, and that most common topologies seek a much smaller diameter. It can be shown that in packet-switched networks, each hop incurs considerable delay due to routing overhead, which is a reason that students should care about the issue.

Latency: In architecture, latency comes in many forms. Extending the idea to message passing is fairly obvious. What is less obvious is how much of it is due to the software protocol stack. Thus, specialized interfaces and routers can be used to reduce latency as a system scales up. The concept can also be covered in an Internet course, observing round-trip times over different numbers of hops.

Bandwidth: It is fairly obvious that data can be transferred at different rates over different kinds of links. Most students will have experienced this effect via different wireless links, or comparing wireless to Ethernet, etc. It is really just a matter of formalizing the terminology. In an architecture class or as part of graph theory the idea of bisection bandwidth can also be introduced with respect to network topology.

Memory Hierarchy:

Cache organization: At the level of an architecture class, once caching has been covered, as soon as bus-based multiprocessing is introduced, the issue of coherency of shared data arises. In an advanced architecture class, a basic snooping protocol, such as SI can be shown to achieve coherency. After a few examples, it becomes clear that such a simple protocol results in excessive coherency traffic, and this motivates the value of a protocol with more states, such as MSI, MESI, or MOESI, which can be briefly described.

Floating-point representation:

Precision: It is easy to explain that higher precision floating point involves a lengthier calculation, moving more bits to and from memory, and that an array of double precision values occupies twice as much memory as single precision. As a result, there is a tradeoff between precision and performance.

Cycles per instruction (CPI): At one level, once the idea of clock cycle has been explained, and the fact that instructions can take different numbers of cycles to execute, it is easy to add the notion that processors can execute fewer or more than one instruction in each cycle. It can be noted that CPI is the inverse of IPC, and that these can be biased by the instruction mix resulting from compiler optimizations (CPI is affected by instruction scheduling). When pipelining is introduced, the CPI calculation process can be demonstrated via a simple by-hand demonstration of a few instructions passing through. In concept it is easy to imagine that superscalar issue greatly affects CPI, and students should be aware that the benefit is less than what they may expect for normal code sequences.

Benchmarks: Students can be shown that most ad-hoc metrics are poor indicators of performance. For example, a processor with a high clock rate that delivers less performance than one with a lower rate (because of other architectural advantages). Thus, benchmark programs are a better indicator of actual performance. But then it is a question of how to define a good benchmark. One kind of program doesn't predict the behavior of another kind. So a suite of benchmarks helps to broaden coverage. But because a suite is an artificial assemblage of programs, it is inherently biased, and so different suites are needed to represent different kinds of workloads.

SPEC mark: Explain differences between arithmetic, geometric, harmonic, and weighted means. Have students explore their values when applied to different data sets, including one with one or two outliers that are much greater than the rest. Notice the excessive impact that the outliers have on the arithmetic and geometric means. Look at the SPEC results for some machines and notice that most reports have one or two outliers. Recompute the mean using harmonic mean, and omitting the outliers. Careful selection of the reports can show two machines trading places in ranking when outliers are ignored.

Peak performance: Use published parameters for an architecture to compute peak performance in MIPS or FLOPS, then see how this compares with benchmark execution reports.

MIPS/FLOPS: Define these terms.

Sustained performance: Define sustained performance, and show some examples in comparison to peak.

10.2 Programming

Parallel Programming paradigms and Notations:

By the target machine model:

SIMD: Discuss operating on multiple data elements at a time with 1 operation/instruction, using a simple example (e.g., array add) with F90 syntax, as a loop, etc.

Microprocessor vector extensions: Introduce (or revisit) SIMD parallelism, its pros and cons, give examples in modern microprocessors (SSE, Altivec), and, if possible, have the students experiment with writing simple programs that use SSE.

Shared memory: Examples of thread programs with both array and control parallelism, with locks and synchronization ops, explain that threads may run in parallel in same address space unless prevented from doing so explicitly, definitely programming projects w/threads (Java, pthreads, etc.)

Shared memory notations: Introduce various ways of parallel programming: (1) Parallel languages, which come in very diverse flavors, for example, UPC, Cilk, X10, Erlang. (2) Extensions to existing languages via compiler directives or pragmas, such as OpenMP. (3) Parallel libraries, such as MPI, Pthreads, Pfunc, TBB, (4) Frameworks such as CUDA, OpenCL, etc., which may incorporate elements of all three. If possible, students should write simple parallel programs to implement the same algorithm using as many of the above four notations as time and resources permit.

compiler directives/pragmas: Introduce the basic directives for writing parallel loops, concurrent sections, and parallel tasks using OpenMP. Have the students write simple OpenMP programs.

libraries: The students should be taught how to write parallel programs using a standard language such as C or C++ and a parallel programming library. Depending on the instructor's preference, any library such as Pthreads, Pfunc, TBB, or TPL can be used. An advantage of Pfunc in an educational setting is that it is open source with a fairly unrestricted BSD-type license and advanced/adventurous students can look at or play with the source. It is designed to permit effortless experimentation w/ different scheduling policies and other queue attributes.

Distributed memory: Example of message passing programs that each process has its own address space with one or more threads, only share data via messages

- Client Server: Java RMI or sockets or web services example, notion of invoking a service in another (server) process, and that client and server may run concurrently

Hybrid: Idea of a single parallel program, with each process maybe running on different hardware (CPU, GPU, other co-processor), and that can be client/server, or MIMD program, or something else

By the control statements:

Task/thread spawning: Thread program examples (Java, pthreads, Cilk), with threads creating and joining with other threads, synchronization, locks, etc.

SPMD: Same code, different data, usually in different processes, so with message passing, but also a style of thread programming, need to trace an example with at least 2 threads/processes to see that each one can take a different path through the program

SPMD notations: Introduce/revisit the SPMD model. The students should be taught about programming in a parallel environment where data-access is highly nonuniform. Introduce/reintroduce the notion and importance of locality. Introduce BSP. Introduce/revisit data movement costs and the distinction between costs due to latency and bandwidth. Given examples of (and, if possible, a brief introduction to a select) languages/frameworks, such as MPI, CUDA, etc., which can used to programming in SPMD model.

Data parallel: Example thread and/or message passing programs, SPMD, SIMD, or just shared memory with parallel loops, operating on elements of a large array or other simple data structure

Parallel loop: Examples of data dependences, and that a parallel loop doesn't have any across loop iterations, show that these are typically data parallel, but whole iterations can run concurrently, example in Fortran or C or whatever of a DO-ALL, and maybe a DO-ACROSS

Tools to detect concurrency defects: e.g., Spin, Intel's Parallel Studio/Inspector

Performance issues:

Computation: Simple example tracing parallel loop execution, and how different iterations can take different amounts of time, to motivate scheduling (static or dynamic)

Computation decomposition strategies: There are standard strategies for parallelizing a computation and its data for parallel execution

Owner's compute rule: An example of one decomposition method - assign loop iterations based on which process/thread owns the data for the iteration

Load balancing: What is performance determined by in a parallel program? When all threads/processes finish, so best when all finish at same time. Introduce idea of balancing statically and/or dynamically, and when dynamic might be needed (missing info at decomposition time)

10.3 Algorithms

Parallel and Distributed Models and Complexity:

Costs of computation:

Asymptotics: See learning outcome for this topic.

time: (1) Review the notion of O(f(n)) asymptotic time complexity of an algorithm, where n is (somehow) related to problem size (e.g., number of elements to be sorted, side-length of a matrix). (2) Adapt the notion to the parallel context by expressing parallel time complexity as O(g(n,p)), where g depends on problem size n and number of cores/processors p. (3) Emphasize that the run time must include the cost of operations, memory access (with possible contention in shared-memory parallel case), and communication (in the distributed-memory parallel case). (4) Introduce parallel speedup and cost-optimality: a parallel algorithm is asymptotically cost optimal if the product of p (the number of cores/processors) and the parallel run time = O(serial run time).

space: Review serial space bound, introduce the notion of parallel space complexity and space optimality, i.e., when the product of p (the number of cores/processors) and the parallel space is of the same order as serial space.

speedup: Introduce and formally define the notion of speedup. Give a simple example, say, by adding n numbers in O(log n) time in parallel with p = n/2. Relate to cost optimality. Present Brent's Theorem to illustrate limits to parallelization: problems usually have inherently sequential portions. (Come back to this when dependencies are covered.)

Scalability in algorithms and architectures: Revisiting the (adding n numbers) example, show that speedups higher than O(n/log n) can be obtained when $p << n$. Use the example to show that speedup depends on both n and p; e.g., here, speedup = np/(n + plog p).

Introduce the notion of efficiency = speedup/p or conceptually, the amount of useful/effective work performed per core. Show that efficiency typically drops as p is increased for a fixed n, but can be regained by increasing n as well. Introduce Amdahl's law.

Model-based notions:

Notions from complexity-theory:

PRAM: (i) Introduce PRAM model, highlighting unrealistic assumptions of O(1)-time shared memory access as well as arithmetic and logical operation and global clock synchronizing each step (SIMD architecture). Introduce EREW, CREW, and CRCW (Common, Arbitrary and Priority) versions for dealing with read-write conflicts; (ii) Illustrate PRAMs' functioning and capability with simple Boolean operations over n bits (OR, AND, NAND, etc.): O(1) time with short circuit evaluation on a common CRCW model vs. O(logn) using a reduction tree on an EREW; show pseudo-codes. Demonstrate that the simple PRAM model empowers one to explore how much concurrency is available in a problem for purely computational reasons --- when not burdened with memory access and synchronization costs. Illustrate by example how unrealistically large PRAMs can be emulated by real parallel computers as a vehicle for obtaining feasible parallel algorithms.

BSP/CILK: Introduce BSP highlighting iterative computation wherein multiple processors compute independent subtasks, followed by periodic global synchronizations that allow processors to intercommunicate. The latency of the underlying network is therefore exposed during the communication/synchronization step (which is ignored in PRAM model). Can illustrate with Boolean OR/AND over n bits or sum/max over n integers resulting in \Omega(n/p + logp) time using p processors. Illustrate by example the use of parallel slack (see the last sentence in the PRAM paragraph).

Notions from scheduling: Take a simple problem such as maximization or summing an array of n integers, and illustrate how the problem can be partitioned into smaller tasks (over subarrays), solved, and then combined (using a task graph structured as a reduction tree or as a centralized "hub-spoke" tree [a/k/a "star"], with all local sums updating a global sum). Use this to illustrate the task graph and the dependencies among parent and child tasks. Alternatively --- or additionally --- consider the floating point sum of two real values, and show its control parallel decomposition into a pipeline. Use this to illustrate task graphs and data dependencies between stages of the pipeline. In either example, calculate the total operation count over all the tasks (work), and identify the critical path determining the lower bound on the parallel time (span).

dependencies: Illustrate data dependencies as above; Mention that handshake synchronization is needed between the producer task and consumer task.

task graphs: Show how to draw task graphs that model dependencies. Demonstrate scheduling among processors when there are fewer processors than the available amount of parallelism at a given level of task graph; illustrate processor reuse from level to level.

work: Calculate work for given task graph using big-O notation.

(make)span: Demonstrate how to identify critical paths in a task graph and calculate a lower bound on parallel time (possibly using big-omega notation). Mention Brent's Theorem, which is based on the critical-path notion. Give examples (e.g., solving a triangular linear system or performing Gaussian elimination).

Algorithmic Paradigms:

Divide & conquer (parallel aspects): Introduce simple serial algorithms, such as mergesort and/or numerical integration via Simpson's Rule or the Trapezoid Rule. Illustrate Strassen's matrix-multiply algorithm via the simple recursive formulation of matrix multiplication. Show how to obtain parallel algorithms using the divide-and-conquer technique. For Strassen, this should be done after teaching parallel versions of usual algorithm (Cannon or Scalapack outer product).

Recursion (parallel aspects): Introduce simple recursive algorithm for DFS. Show how a parallel formulation can be obtained by changing recursive calls to spawning parallel tasks. Consider the drawback of this simple parallel formulation; i.e., increased need for stack space.

Series-parallel composition: Illustrate that this pattern is the natural way to solve many problems that need more than one phase/sub-algorithm due to data dependencies. Present one or more examples such as (i) time-series evolution of temperature (or your favorite time-stepped simulation) in a linear or 2D grid (each time step, each grid is computed as the average of itself and its neighbors), (ii) O(n)-time odd-even transposition sort, or (iii) O(1)-time max-finding on a CRCW PRAM (composition of phases comprising all-to-all comparisons followed by row ANDs followed by identification of the overall winner and output of the max value). It would be valuable to show the task graph and identify the critical path as the composition of individual critical paths of the constituent phases. A connection with CILK would be a valuable to expose both top illustrate a practical use and to establish nonobvious connections.

Algorithmic problems:

Communication:

Broadcast: Introduce simple recursive doubling for one-to-all and all-to-all among p processes in log p steps. More advanced efficient broadcast algorithms for large messages could also be taught after covering gather, scatter, etc. For example, one-to-all broadcast = scatter + allgather. Also pipelined broadcast for large messages (split into packets and route along same route or along disjoint paths).

scatter/gather: See above.

Asynchrony: Define asynchronous events and give examples in shared- and distributed-memory contexts.

Synchronization: Define atomic operations, mutual exclusion, barrier synchronization, etc., examples of these and ways of implementing these. Define race conditions with at least one example and show how to rewrite the code to avoid the race condition in the example.

Sorting : (i) Explain the parallelization of mergesort wherein each level starting from bottom to top can be merged in parallel using n/2 processors thus requiring O(2+ 4+ … + n/4 + n/2 + n) = O(n) time. Using p<=n/2 processors will lead to O(n/plog(n/p) + n) time, hence p=log n is a cost-optimal choice. (ii) Highlight that a barrier (or a lock/Boolean flag per internal node of the recursion tree) on shared memory machine or messages from children processors to parent processors in a local memory machine would be needed to enforce data dependency; (iii) Mention that faster merging of two n/2 size subarray is possible, e.g., in O(log n) time on a CREW PRAM using simultaneous binary search using n processor, thus yielding O(log^2n)-time algorithm.

Selection: (i) mention that min/max are special cases of selection problem and take logarithmic time using a reduction tree; (ii) for general case, sorting (e.g., parallel mergesort) is a solution.

Graph algorithms: Basic parallel algorithms for DFS and BFS. Preferably include deriving expressions for time, space , and speedup requirements (in terms of n and p). Parallel formulations and analyses of Dijkstra's single-source and Floyd's all-source shortest path algorithms.

Specialized computations: Example problem - matrix multiplication (AxB = C, nxn square matrices): (i) Explain the n^3-processor O(logn)-time PRAM CREW algorithm highlighting the amount of parallelism; this yields cost optimality by reducing processors p in O(n^3/logn) ensuring O(n^3/p) time (exercise?). (ii) Explain that a practical shared-memory, statically mapped (cyclic or block) algorithm can be derived for p <= n^2 by computing n^2/p entries of product matrix C in a data independent manner; (iii) For p<= n, the scheduling simplifies to mapping rows or columns of C to processors; mention that memory contention can be reduced by starting calculation at the ith column in the ith row (exercise?). (iii) For a local memory machine with n processors with a cyclic connection, the last approach yields a simple algorithm by distributing the ith row of A and the ith column of B to P_i, and rotating B's columns (row-column algorithm) - yields O(n^2) computation and communication time; mention that for p<n, row and column bands of A and B can be employed - derive O(n^3/p) time (exercise?). (iv) For 2-D mesh, explain cannon's algorithm (may explain as refinement of n^2-processor shared-memory algorithm, wherein each element is a block matrix).

Termination detection: Define the termination detection problem

- simple message based termination detection:
- single pass ring termination detection algorithm
- double pass ring termination detection algorithm
- Dijkstra-Scholten algorithm
- Huang algorithm

Leader election/symmetry breaking: define the Leader election problem

- Leader election in a ring:
    - Chang and Roberts algorithm
- General, ID based leader election:
    - Bully algorithm

10.4 Crosscutting and Advanced Topics

Why and what is parallel/distributed computing?: examples: multicores, grid, cloud, etc.

Crosscutting topics: can be covered briefly and then highlighted in various contexts

Concurrency: The notion of inherent parallelism can be illustrated by a high level specification of the process to achieve the desired goal. A simple example to consider is sorting – quick-sort or merge-sort. An important idea to illustrate with respect to inherent parallelism is how the level of abstraction in the specification affects the exposed parallelism -- that is, illustrating how some of the inherent parallelism may be obscured by the way the programmer approaches the problem solution and the constructs provided by the programming language. Another important idea to illustrate is that of nesting – a higher level step may itself allow exploitation of parallelism at a finer grain. A yet another important idea is the need to weigh the available parallelism against the overhead involved in exploiting it.

Non-determinism: Non-determinism is an inherent property when dealing with parallel and distributed computing. It can be easily illustrated by discussing real-life examples where, e.g., different runs of a parallel job give different answers due to non-determinism of floating-point addition. The dangers of this can be illustrating by talking about order of operations and the need for synchronization to avoid undesirable results.

Locality: The performance advantages of locality are easy to explain and can be illustrated by taking examples from a wide spectrum of data access scenarios. This includes cache data locality in the programming context, memory locality in paging context, disk access locality, locality in the context of virtualization and cloud computing, etc. Both spatial and temporal aspects of locality must be clarified by illustrating situations

where only or both may be present. Simple eviction/prefetching policies to take advantage of locality should also be illustrated with examples. Relationship of temporal locality to the notion of working set should also be explained.

Power consumption: Power consumption of IT equipment is a topic of increasing importance. Some general principles of power savings such as use of sleep states and reduced voltage/frequency operation can be introduced along with its impacts on power consumption, performance and responsiveness. It is also important to make a distinction between reducing power vs. reducing energy consumption by using simple examples. Finally, this topic provides a perfect opportunity to discuss the role of user behavior and behavior change in truly reducing IT energy consumption.

Fault tolerance: Fault tolerance is a fundamental requirement to ensure robustness and becomes increasingly important as the size of the systems increases. In a system composed of a large number of hardware elements (e.g., processing cores) or software element (e.g., tasks), the failure of a few is almost a given, but this should not disrupt or silently corrupt the overall functioning. Some important aspects to cover include: an introduction to the increasing need for fault-tolerance illustrated by simple equations, a brief classification of faults (transient, stuck-at, byzantine, …), and illustration of some basic techniques to deal with them (retry, coding, replication and voting, etc.).

Performance modeling: Performance is a fundamental issue at all levels of computing and communications, and thus needs to be addressed in most topics, including architecture, programming, and algorithms. Indeed, performance topics appears in all of these topics. In addition, it is important for students to learn basic techniques for analyzing the performance impact of contention for shared resources. The basic concepts include the idea of a queuing system, infinite server vs. single server queues, stability of queuing systems, utilization law, Little's law, open and closed networks of resources and applying Little's and utilization laws, memoryless behavior, and simple M/M/c queuing analysis. The ideas can be illustrated with examples from architecture, networks, and systems.

Current/Hot/Advanced Topics:

Cluster Computing: A cluster is characterized by a set of largely homogeneous nodes connected with a fast interconnect and managed as a single entity for the purposes of scheduling and running parallel and distributed programs. Both shared memory and message passing alternatives can be briefly discussed along with their pros and cons. Cluster computing can be illustrated by using the specific example of a Beowulf cluster, and briefly discussing the use of MPI and MapReduce paradigms for solving a simple distributed computing problem.

Cloud/Grid Computing: The notion of virtualization is crucial for understanding cloud computing and should be briefly covered, along with structure of some popular virtualization software such as VMware and Xen. Cloud computing could then be introduced assisted by machine, network, and storage virtualization. A brief discussion of VM scheduling and migration is essential to provide an overview of how cloud computing works. Cloud storage and cloud services concepts can be easily illustrated by using the example of drop-box – a popular cloud storage service. Alternately (or in addition), a hands on demonstration of how resources can be requested and used on a commercial platform such as Amazon EC2 is very useful introducing cloud computing to the students. Grid computing can be briefly introduced along with a brief mention of Globus toolkit.

Peer to Peer Computing: The students are likely to have already made a good deal of use of available P2P services such as Bit Torrent and those could be used as a starting point of discussion of P2P. The important concepts to get across in P2P are: (a) the notion of give and take in cooperative P2P services, (b) structured vs. unstructured content organization and searches, and (c) pros and cons of P2P vs. client-server based implementations of services. The structure of Bit Torrent, and the key concepts of file segmentation, seed, leecher, tit-for-tat, and chocking should be explained briefly. Skype can be introduced as another form of P2P application.

Distributed Transactions: Consistency maintenance in the face of concurrent updates is a crucial learning outcome that can be illustrated with a simple database update example. The need for both strict consistency and looser forms of consistency should be illustrated by using appropriate examples (e.g., banking vs. web browsing). The notions of optimistic and pessimistic concurrency control can be illustrated by a simple example. Consistency in the presence of multiple copies is a somewhat more advanced topic that can be introduced briefly.

Security and privacy: Security and privacy concerns multiply both with an increase in the size of the systems (in terms of number of independent agents and information repositories), and an increase in intelligence (which requires that more detailed information be learnt and shared). Example to illustrate the risks can be drawn from social networks, customer data learnt/maintained by Google, Amazon and other prominent companies, or even from emerging areas such as matching supply and demand in a smart grid. The tradeoff between security, privacy, and intelligence of operations can also be illustrated with these examples.

Web searching: Web searching requires a substantial amount of distributed processing and pre-computation in order to provide quick answers. A brief discussion of web-crawling to locate new web pages and update deleted pages, building indexes for quick search, parallel search, dealing with multiple cached copies, and ranking of search results is important to convey the range of activities involved in web-search. The ideas are best illustrated via a concrete example assuming a popular search engine such as Google.

Social networking: Social networking is by now well entrenched and it is likely that most beginning CS/CE students have already used one or more such services. The purpose of this topic is to sensitize the students to ways in which social networking information can be exploited to provide enhanced services that account for social context and other information derived from social networking data. The tradeoff between usability and privacy can also be illustrated using these examples.

Collaborative computing: Collaborative computing refers to active involvement of multiple users (or devices) to accomplish some objective. Examples of collaborative computing include shared document editing (e.g., Google docs), multiplayer games, and collaboration between enterprises. Some of these applications can be discussed briefly along with some important distributed systems concepts (e.g., consistency and synchronization) applied to them.

Web services: Web services form the basis for browser based interaction between users and the enterprise servers that provide the relevant data and functionality. The course can illustrate web-services by a simple programming exercise to fetch say, current stock price using Java or .Net framework. The course should also introduce the basic web-services infrastructure such as publication via UDDI, description of functionality via WSDL, invocation via SOAP RPC, and invocation using XML/SOAP.

Pervasive/Mobile computing: Mobile computing, possibly assisted by cloud computing for offloading heavy-duty computation and intelligent decision-making is emerging as a way to support applications of importance to a community or society at large. Such applications include monitoring and understanding of evolving real-world events such as traffic congestion on highways, unfolding disasters, or social mood. Pervasive computing covers an even larger area that includes ad hoc or embedded devices such as surveillance cameras in malls, occupancy sensors in rooms, seismic sensors, etc. While illustrating these as emerging examples of distributed computing, the unique aspects of these environments can be briefly discussed, e.g., no coupling between devices, ad hoc & changing topologies, large scale, possibility of exploiting context, security, privacy and reliability issues, etc.

11. Appendix III: Sample Elective Course: Introduction to Parallel and Distributed Computing

This single course is designed to cover most of the proposed core topics into one elective parallel and distributed computing course. Preferred adoption model is integration of proposed topics into core level courses. Some samples would be collected and posted at the curriculum site.

3 semester credits or 4 quarter credits.
Total number of hours of in-class instruction = 45.

Prerequisites: Introductory courses in Computing (CS1 and CS2)

Syllabus:

- High-level themes: Why and what is parallel/distributed computing? History, Power, Parallel vs. Distributed, Fault tolerance, Concurrency, non-determinism, locality (2 hours)
- Crosscutting and Broader topics: power, locality; cluster, grid, cloud, p2p, web services (2 hours)
- Architectures (4.5 hours total)
    - Classes (3 hours)
        - Taxonomy
        - Data versus control parallelism: SIMD/Vector, Pipelines, MIMD, Multi-core, Heterogeneous
        - Shared versus distributed memory: SMP (buses), NUMA (Shared Memory), Message passing (no shared memory): Topologies
    - Memory hierarchy, caches (1 hour)
    - Power Issues (1/2 hour)
- Algorithms (17.5 hours total)
    - Parallel and distributed models and complexity (6.5 hours)
        - Cost of computation and Scalability: Asymptotics, time, cost, work, cost optimality, speedup, efficiency, space, power - (4 hours)
        - Model-based notions: PRAM model, BSP - (1 hour)
        - Notions from scheduling: Dependencies, task graphs, work, makespan – (1.5 hours)
    - Algorithmic Paradigms (3 hours)
        - Divide and conquer, Recursion
        - Series-parallel composition

- Algorithmic Problems – (8 hours)
  - Communication: broadcast, multicast, reduction, parallel prefix, scatter/gather (2 hours)
  - Synchronization: atomic operations, mutual exclusion, barrier synchronization; race condition (1 hour)
  - Specialized computations: Rrepresentative sample from among matrix product, transposition, convolution, and linear systems (3 hours)
  - Sorting, selection (2 hour)
- Programming (19 hours total)
  - Parallel Programming paradigms – (3 hours)
    - By the target machine model: Shared memory, Distributed Memory, Client-Server, Hybrid - (1.5 hours)
    - By the control statements: Task/thread spawning, SPMD, Data parallel, Parallel loop – (1.5 hours)
  - Parallel programming notations – (8=6+2 hours)
    - Shared memory notations: language extensions, compiler directives/pragma, libraries
    - SPMD notations: MPI, CUDA, etc.
  - Semantics and correctness issues (4 hours)
    - Synchronization: shared memory programs with critical regions, producer- consumer; mechanism for concurrency (monitors, semaphores, etc.)
    - Concurrency defects: deadlock (detection, prevention), race conditions (definition), determinacy/indeterminacy in parallel programs
  - Performance issues (3 hour)
    - Computation: static and dynamic scheduling, mapping and impact of load balancing on performance
    - Data: Distribution, Layout, and Locality, False sharing, Data transfer (1 hour)
    - Performance metrics: speedup, efficiency, work, cost; Amdahl's law; scalability
  - Tools (1 hour)
    - Debuggers (1 hour)
    - Performance monitoring (1 hour)