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

12/23/2010

Chtchelkanova, Almadena (NSF), Das, Sajal (University of Texas at Arlington, NSF), Das, Chita (Penn State, 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, NSF), Prasad, Sushil (Georgia State University), Prasanna, Viktor (University of Southern California), Robert, Yves (INRIA, France), Rosenberg, Arnold (Colorado State 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)

Abstract: This is the preliminary version of core topics in parallel and distributed computing that a student graduating with a Bachelors degree in Computer Science or Computer Engineering is expected to have covered. Additional elective topics are expected. This is expected to engage the various stakeholders for their feedback and early adoption. This document contains an introductory write up on curriculum's need and rationale, followed by the proposed topics, level of coverage, and learning outcomes, and additional material in the appendix on suggestions on how to teach individual topics, a cross-reference matrix on core courses vs. topics, and a sample course. We are seeking early adopters of the curriculum for winter/spring terms of 2011 in order to get a preliminary evaluation of our proposal. Email us your brief plans for integrating and evaluating some of the proposed topics into your core/elective course, or if you already teach such an integrated course. Some seed level funds are available for early adopters with support from NSF and Intel.

Contact: Sushil Prasad (sprasad@gsu.edu)

Table of Contents

1. Introduction

Parallel and Distributed Computing (PDC) now permeates most computing activities - the "conscious" ones, in which a person works explicitly on programming a computing device, and the "unconscious" 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 "conscious" and "unconscious" 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 home and office PCs and laptops has empowered even common users to become contributors to advanced computing technology. 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 or integrating parallelism in 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 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, 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 has continued through Dec 2010.

The goal of the ongoing meetings has been 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, recommending 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 the implication 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, in 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 as a 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 must 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 historical significance or promote them to more advanced courses.

The results of our preliminary deliberations are contained in this document. In the three PDC sub-areas of Programming, Algorithms, and Architecture, 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[1] using the following notation[2]
   K= Know the term (basic literacy)
   C = Comprehend so as to paraphrase/illustrate
   A = Apply it in some way (requires operational command)

[1] (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.

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

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.

Afterwards, it was felt that some higher-level themes, crosscutting topics and current/advanced topics should also be considered, which could not accommodated within the three areas. This led to a fourth table as well.

This preliminary version of the proposed core curriculum is being released in Dec 2010. We are seeking early adopters of the curriculum for winter/spring terms of 2011 (and later) in order to get a preliminary evaluation of our proposal. These adopters will include (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.

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 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. 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 overview the reasons for developing a PDC curriculum at this time (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 for its underlying intent (Section 3). Sections 4, 5, and 6 then continues with rationale for each of the three major topic areas in the proposal: architecture, programming, and algorithms. The proposed curriculum is contained in Section 7. For the benefit of instructors, Appendix I contains 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 six 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. 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 one might call *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.

Another branch, related to the hydra-like *shared-memory computers,* had multiple memory boxes, as well as multiple processors. For efficiency, certain processors had preferential 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 "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 multithreading, pipelining, superscalar issue, and short-vector instructions. All of this heterogeneous parallelism is now wrapped into common 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 becoming difficult to ignore parallel computing in even the core programming curriculum.

The most obvious place in which parallelism is exposed is the graphical user interface (GUI), where asynchronous events and multiple threads must be managed for any user interaction that rises above the sophistication of a command-line interface. Threads appear also in libraries of data structures, such as the C++ Standard Template Library and the Java Collections Framework, that are increasingly being used in the teaching of data structures.

In the past, it was possible to relegate issues regarding parallelism 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 it is clear that 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 earlier stages of an undergraduate career.

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. The evolution just described has made this challenge even more daunting. 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, with allowance for local variations in emphasis. The field is changing too rapidly for a 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 thus see these topics as being most appropriately sprinkled throughout a standard curriculum in a way that enhances what is already taught, and brings parallel and distributed computing into relationship with existing material in whatever ways are most natural. Relegating these subjects to a separate course is not the best means to shift the mindset of students away from purely sequential thinking. However, it may be that such an arrangement works better for some departments.

3. How to Read this Proposal?

When reading this proposal, it is essential to keep in mind that many of the topics discussed have multiple levels of potential exposition in the broader curriculum. Upon seeing a topic mentioned, one should not jump to any immediate conclusion regarding its suitability for coverage with undergraduates. 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 developing parallel and distributed thinking in students.

For each of the topics we discuss, we have provided one or more examples of where and how it can be covered. These should not be taken as prescriptive: We are not saying that this is the preferred form of coverage. Instead, the examples are intended to illustrate one possibility for coverage, with the goal of initiating thought that leads to alternate possibilities.

We do not envision the curriculum as being implemented like a checklist, where each item is addressed at precisely one point. A more effective use of this curriculum guideline is to encourage instructors to find as many ways as appropriate to insert coverage of PDC 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 professors are always conscious of the implications of parallelism and distribution of computation with respect to each topic in their courses.

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 three sections we present rationales for the three areas of computer science and engineering into which we have divided the learning goals of the proposed curriculum. Because the current trend is driven by architectural changes in response to fundamental issues of physics, we begin with the rationale for that area, as it establishes certain foundations for the way that parallelism and distribution are expressed in the other areas. At a pragmatic level, students encounter these topics first in high-level languages, so we

present the rationale for programming topics next. The third area into which the learning goals are grouped encompasses algorithms, and its rationale appears last.

4. Rationale for Architecture Topics

Existing computer science and engineering curricula generally include the topic of computer architecture to various degrees. This may range from a specific required course in the subject to a distributed set of concepts that are addressed across multiple introductory courses.

As an example of the latter, consider that in an early programming course the instructor may introduce parameter passing or linked data structures by explaining the idea that computer memory is divided into unit cells of a fixed size, 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 instructor's explanation of the functionality of a programming language construct. Readers can no doubt think of many other instances in which architecture is blended into early course content, without relying on its coverage in a prerequisite architecture course.

While the topics of parallel architecture in this proposed curriculum could be gathered into an upper level course and given explicit treatment, they also have the potential to be interwoven in lower-level courses in much the same way that concepts of serial architectures now appear in those courses. Indeed, as the bulk of laptop and desktop systems shift to multicore, multithreaded designs with vector extensions, and more languages and algorithms move to support data and thread parallelism, 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 wireless 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 will go hand in hand with adding topics related to programming and algorithms. Because, even though languages and algorithms can be developed around abstract models of parallelism, those that have practical value will bear a relationship to what happens in hardware. Consequently, explaining the reasoning behind a language construct, or why one algorithmic approach is chosen over another in practice, will involve a connection with architecture.

A shift to "thinking in parallel" has often been described as a requisite for the transition to widespread use of parallelism. The architecture curriculum extensions described here anticipate that this shift will be holistic in nature, and that many of the fundamental concepts of parallelism and distribution will be interwoven among traditional ideas in the teaching of computer science and engineering. In addition, just as there are advanced programming, algorithms, and architecture courses in today's curricula, it is expected that more advanced topics from PDC will find their places in those or similar courses.

The topics chosen are meant to organize the principles that underlie the different directions in which hardware and the corresponding programming models are moving. There are many architecture topics that could be included, but the goal is to identify those that most directly impact and inform all undergraduates in computer science and engineering, and which are well established and likely to remain significant over time.

For example, the use of graphics processors for general purpose computing is presently a hot topic, but there is no way of knowing whether it will continue to be of interest. On the other hand, multithreading and vector parallelism have been with us for over four decades, and are likely to remain among the common computing models because they naturally arise from properties of physical locality and control signal distribution in the hardware. An instructor could choose to present these fundamental topics in the context of graphics processing in the near term, but even if the GPGPU falls from favor in a few years, students will still need to be aware of threads and vectors.

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 is the largest collection among these topics, and is meant to encourage a survey of 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 broken up along two taxonomic dimensions: parallelism of control vs. data, and the degree to which memory is partitioned.

Highly data parallel models that students should be aware of include superscalar, pipeline, stream, vector, and dataflow. Shifting more toward control parallelism, models of multithreading, multiprocessing, multicore, cluster, and grid/cloud are important. All of these, of course, appear in heterogeneous mixtures in modern systems.

At a logical level, parallel models distinguish between communicating through memory and over networks. Each of these can be further developed in greater detail, and the two blend together where latencies are too great to be ignored, but not so great as to demand an alternative algorithmic paradigm or programming model.

Memory Hierarchy is another topic that is covered in the traditional curriculum, but when parallelism and distribution come into play, the issues of atomicity, consistency, and coherence become more significant, to the point of affecting the programming paradigm, in order to obtain optimized performance.

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. In the teaching examples, however, some are also described in terms of how coverage may be accomplished in lower level courses. The architecture area also includes a set of topics that are described as belonging to a second course in architecture or other advanced courses. Clearly, these are not meant to be part of a core curriculum. Rather, 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 four subtopics: Paradigms, 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, it is clear that 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 in conjunction as necessary and appropriate.

Paradigms: 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: *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)/Altivec macros, some database operations, some operations in data structure libraries, and the languages used for early vector machines. *Shared-memory* is the paradigm of OpenMP and Intel's Thread Building Blocks. *Distributed memory* is the paradigm underlying MPI. And hybrid is when any of the previous three paradigms co-exist in a single program. The 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 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 in the first classification. The same is true of programs following the data parallel model. The task spawning model can work with 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.

Notations: 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 group 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. 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 ideas 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 with a firm background in the conceptual underpinnings of parallel and distributed computing (PDC). Not only is parallelism becoming pervasive in computing, but technology is changing in this dynamic field also at a rapid pace. Students whose education is tied too closely to obsolescent technology will be at a disadvantage.

Paradigm shifts in the computing field are not new. To mention just a few instances from the past: (1) The VLSI revolution of the late 1970s and early 1980s enabled 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 from the details of the underlying platform architecture and provide the basis for developing applications on current and 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 technology.

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 have 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; the series-parallel paradigm that one encounters, e.g., in multi-threaded computing environments; etc.

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. Proposed Curriculum

7.1 Notations

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
DM	Discrete Structures/Math

ADVANCED/ELECTIVE COURSES:

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

Note: Interpretation of suggested number of hours needs to be done carefully. For example, students need to be able to achieve "A" level expertise in shared memory programming. The numbers of hours required for this should not be interpreted simply what has been assigned to the topic "shared memory" under "target machine model." Rather, it is the total of all hours assigned for this topic as well as several relevant topics such as SPMD, tasks and threads, synchronization, etc.

7.2 Architecture Topics

Table 1: Architecture

Topics	BLOOM #	HOURS	Where Covered	Learning Outcome
Classes				
Taxonomy				Architectural classes, Flynn's taxonomy, data vs. control parallelism, shared/distributed memory machines
Data vs. control parallelism				
Superscalar (ILP)	K	0.25 to 3, depending on level	Systems	Describe uses of multiple instruction issue and execution (different instructions on different data)
SIMD/Vector (e.g., SSE, Cray)	K	0.5 to 1	Systems	Describe uses of SIMD/Vector (same operation on multiple data items)
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, Arch 2	
• OoO execution	N		Arch 2	
Streams (e.g., GPU)	K	0.2 to 1	Systems	Know that stream-based architecture exists in GPUs for graphics
Dataflow	N		Arch 2	
MIMD	K	0.1 to 1	Systems	Identify MIMD instances in practice (multicore, cluster, e.g.)
Simultaneous Multi-Threading (e.g.,	K	0.2 to 1.5	Systems	Distinguish SMT from multicore (based on which resources are shared); e.g. Intel's hyper-threading is an implementation of SMT

Hyper-threading)				
Highly Multithreaded (e.g., MTA)	N		Arch 2	
Multicore	C	0.5 to 1	Systems	Describe how cores share resources (cache, memory) and resolve conflicts
Heterogeneous (e.g., Cell)	K		Systems	Recognize that multicore may not all be the same core.
Shared vs. distributed memory				
SMP				UMA architecture
• Buses	C	0.5 to 2	Systems	Single resource, limited bandwidth and latency, snooping, scalability issues
NUMA (Shared Memory)	N		Arch 2 (K)	Physical limitations of latency and bandwidth results in such architectures
• CC-NUMA	N			
• Directory-based CC-NUMA	N			
Message passing (no shared memory)				Shared memory architecture breaks down when scaled due to physical limitations (latency, bandwidth) and results in message passing architectures
• Topologies	C	0.5 to 1.5	Systems	Various topologies - linear, ring, mesh/torus, tree, hypercube, clique, crossbar
• Diameter	C	0.1 to 0.5	Systems	Appreciate differences in diameters of various topologies
• Latency	K	0.2 to 1	Systems	Know the concept, implications of scaling
• Bandwidth	K	0.1 to 0.5	Systems	Know the concept
• Circuit switching	N			
• Packet switching	N			
• Routing	N			
Memory Hierarchy				
• Cache organization	C		Systems	Know the cache hierarchies, shared caches (as opposed to private caches) result in coherency and performance issues for software
• Atomicity	N		Arch 2	Should be covered in programming

• Consistency	N		Arch 2	
• Coherence	N		Arch 2	Describe how cores share cache and resolve conflicts - may be covered in software/programming
• False sharing	N		Arch2/ Par Prog	Awareness, origin
• Impact on software	N		Arch2/ Par Prog	Issues of cache lines length, memory blocks,
Floating point representation				These topics are supposed to be in ACM/IEEE core curriculum already – included here to emphasize their importance, especially in the context of PDC.
Range	K		CS1/CS2/Systems	
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	
Error propagation	N		Arch 2	
754 standard	K		Systems	
Performance metrics				
Cycles per instruction (CPI)	C	0.5 to 1.5	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
• Spec mark	K	0.5 to 1	Systems	Aware of pitfalls in relying on averages
• Bandwidth benchmarks	N			
Peak performance	C	0.5 to 1	Systems	Understanding peek performance, how these may not be applicable for estimating real performance, illustrate fallacies
• MIPS/FLOPS	C	0.1 to 0.5	Systems	Understand
Sustained performance	C	0.1 to 0.5	Systems	Know difference between peak and sustained performance, how to define, measure, different benchmarks
• LinPack	N			

7.3 Programming Topics

Note: The sub-topics in lighter shade were collectively dealt with and folded into higher-level topics with corresponding outcomes. The sub-topics have been retained to (i) enable completing Bloom classification, courses, etc., and (ii) serve for posterity - even those marked N will be useful for non-core elective classes.

Table 2 Programming

Topics	BLOOM#	HOURS	Where Covered	Learning Outcome
Parallel Programming paradigms				
By the target machine model				
SIMD	K	1/2	DS/A; Systems	Understand common vector operations (e.g. SSE/Altivec macros, Fortran-90 array operations)
Shared memory	A	1	CS2, DS/A	Be able to write correct thread- based programs (protecting shared data) and understand how to obtain speed up
Distributed memory	C	1	DS/A; Systems	Know basic notions of messaging among processes, different ways of message passing, collective operations
Client Server	C	1 1/2	CS2, DS/A	Know notions of invoking and providing services (e.g., RPC, RMI, web services) - understand these as concurrent processes
Hybrid	K	1/2	Systems	Know the notion of programming over multiple classes of machines simultaneously (CPU, GPU, etc.)
By the control statements				
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, CILK, JAVA threads, etc.)
SPMD	C	1	CS2; DS/A	Understand how spmd program is written and how it executes

Data parallel	A	1	CS2; DS/A; Lang	Be able to write a correct data-parallel program and get speedup, should do an exercise
Parallel loop	C	1	DS/A; Lang	Know, through an example, one way to implement parallel loops, understand collision/dependencies across iterations (e.g., openMP, Intel's TBB)
Parallel Programming notations				
Array languages				
Microprocessor vector extensions	K	1/4	Systems	Know examples - SSE/Altivec macros
Fortran 90 and successors/C++ array extensions	N			
Shared memory notations				
Language extensions	K	1/2	CS2; DS/A	Know about language extensions, such as spawn/sync, fork/join
Compiler directives/pragmas	C	1	DS/A; Lang	Understand what simple directives mean (parallel loop, concurrent section), show examples
Libraries	C	1	CS2; DS/A; Lang	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.
SPMD notations	C	1	CS2; DS/A	See different examples, know the existence of MPI, CUDA, and some others (such as OpenCL, Global Arrays, BSP library)
CUDA/OpenCL				
MPI				
Global arrays				
BSP				
Functional/logical languages	N			Requires considerable amount of time and experience, but worth covering in an advanced course.
Parallel Haskell	N			
Parlog	N			
Semantics and correctness issues				
Tasks and threads	K	1/2	CS2; DS/A; Systems,	Know the relationship between number of tasks/threads/processes and processors/cores for performance and impact of context switching on performance

			Lang	
Fixed number of virtual processors				
Variable number of virtual processors.				
OpenMP implicit tasks				
Synchronization	A	1 1/2	CS2, DS/A, Systems	Be able to write shared memory programs with critical regions, producer-consumer, and get speedup; know the notions of mechanism for concurrency (monitors, semaphores, … - [from ACM 2008])
Critical regions	A			
Producer-consumer	A			
Monitors	K			
Concurrency defects	C	1	DS/A, Systems	Know the notions of deadlock (detection, prevention), race conditions (definition), determinacy/indeterminacy in parallel programs (e.g., adding floating point numbers in array - the outcome may dependent on the order of execution)
Deadlocks	C			
Unwanted Races	K			
Memory models	N			
• Sequential consistency	N			
• Relaxed consistency	N			
Tools to detect concurrency defects	K	1/2	DS/A, Systems	Know the existence of tools to detect race conditions,
Performance issues				
Computation	C	1 1/2	CS2; DS/A	Understand the basic notions of static and dynamic scheduling, mapping and impact of load balancing on performance
Computation decomposition strategies	C			
• Owner's compute rule	C			
• Decomposition into atomic tasks and then coalescing	C			
• Work stealing	N			

Elementary program transformations	N			
• Loop fusion	N			
• Loop fission	N			
• Skewing	N			
Load balancing	C			
Scheduling and mapping	C			
• Static				
• Dynamic				
Data	K	1	DS/A; Lang	Understand impact of data distribution, layout and locality (what it means, general issues) 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)
Organization of the data	K			
• Data distribution				
• Block				
• Cyclic				
• Block-cyclic				
• Data layout				
Data locality	K			
False sharing	K			
Message aggregation				
Performance monitoring	K	1/2	DS/A	Know of tools
Tools				
Performance metrics	C	1	CS2; DS/A	Know the basic definitions of performance metrics (speedup, efficiency, work, cost), Amdahl's law; know the notions of scalability
Speedup	C			
Efficiency	C			
Redundancy				
Isoefficency				

Amdahl's law	K			
Gustafson's Law	N			

7.4 Algorithm Topics

Table 3 Algorithms

Topics	BLOOM #	HOURS	Where Covered	Learning Outcome
Parallel and Distributed Models and Complexity				Be exposed to the models and to the intrinsic degree of parallelism of some elementary key algorithms (e.g. maximum, prefix)
Costs of computation:				Able to follow arguments for parallel time and space complexity given by instructor
Asymptotics	C		DS/A	Recognizing upper bounds big O vs. lower bounds big Omega, elementary level command over big O. e.g. mergesort is limited by tree depth leading to O(logn) time with unbounded parallelism.
Time	C	3	DS/A	
Space	C	1	DS/A	
Speedup	C	1	DS/A	Use parallelism either to solve same problem faster or to solve larger problem in same time
Cost reduction:	N			
Space compression, etc.	N			
Cost tradeoffs:				
Time vs. space,	N		DS/A	
Power vs. time, etc.	N		DS/A	

Scalability in algorithms and architectures	C/K	1	DS/A	Understand that more processors does not always mean faster execution, e.g. inherent sequentiality of algorithmic structure, DAG representation with a sequential spine
Model-based notions:				
Notions from complexity-theory:				
• PRAM	K	2	DS/A	PRAM is an exemplar of simplest parallel model. Embarrassingly parallel problems lend themselves to easy parallel solutions just by employing many processors.
• BSP/CILK	K	2	DS/A	There are higher-level algorithmic abstractions that encapsulate more aspects of real architectures. Any one of BSP or CILK would be a good option to introduce higher level programming model.
• Simulation/emulation,	N		Algo 2	The high rate of technological change in parallel/distributed platforms demands those students understand programming abstractions and how to apply abstractions to a variety of actual hardware and software architectures.
• P-completeness,	N		Algo 2	
• #P-completeness	N		Algo 2	
• Cellular automata	N		Algo 2	This important model introduces new aspects of parallelism/distributed computing
Notions from scheduling		2		Understand how to decompose a problem into tasks
• Dependencies,	A	1	CS1/CS2, DS/A	
• Task graphs,	C	1	DS/A; SwEngg	This is a concrete, algorithmic abstraction. It is the level at which parallelism is exposed and exploited.
• Work,	K	1	DS/A	
• (Make)span	K	2	DS/A	Makespan is parallel time,
Algorithmic Paradigms				
Divide & conquer (parallel aspects)	C	2	CS2, DS/A	CS2/Data Structures/Algorithms - mergesort - introduce tree and expose parallelism; Higher Level Algorithms (CS7) - may illustrate using Strassen's matrix-multiply; numerical integration --- choose examples
Recursion (parallel aspects)	C	2	CS2, DS/A	recognize algorithms that unfold yielding tree structures have subtrees that can be computed independently, in parallel
Scan (parallel-prefix)	N		ParAlgo	Cover a sampler of Blelloch's examples; present as a "high-level" algorithmic tool
Reduction (map-reduce)	N		Algo 2	Tree structure implicit in scalar product (or mergesort or histogram) can be

			(K/C)	employed for this notion
Stencil-based iteration	N		ParAlgo	Mapping and load balancing illustrated
Dependencies:	K	1	Systems	Impacts of dependencies
"Oblivious" algorithms	N		ParAlgo	
Blocking	N		ParAlgo	This is an algorithmic manifestation of memory hierarchies
Striping	N		ParAlgo	This is an algorithmic manifestation of memory hierarchies
"Out-of-core" algorithms	N		ParAlgo	
Series-parallel composition	C	2	CS2(K), Systems(C)	Abstraction that embodies thread programming
Graph embedding as an algorithmic tool	N		ParAlgo	This material is needed for the same reason as "simulation/emulation"
Algorithmic problems				The important thing here is to emphasize the parallel/distributed aspects of the topic
Communication				
Broadcast	C/A	3	DS/A	Represents method of exchanging information - one-to-all broadcast (by recursive doubling)
Multicast	K/C		DS/A	Illustrate macro-communications on rings, 2D-grids and trees
Scatter/gather	C/A		DS/A	
Gossip	N			
Asynchrony	K	1	CS2	Asynchrony as exhibited on a distributed platform, existence of race conditions
Synchronization	K	1	CS2, DS/A	Aware of methods of controlling race condition
Sorting	C	2	CS2, DS/A	Parallel merge sort
Selection	K	1	CS2, DS/A	Min/max, know that selection can be accomplished by sorting
Graph algorithms:		3		
Search	C			Know how to carry out BFS like parallel search in a graph or solution space
Path selection	N			
Specialized computations	A	5	CS2, DS/A	Choose representative sample from among matrix product, transposition, convolution, and linear systems; Recognize how algorithm design reflects the structure of the computational problems.
Convolutions	Opti			Block or cyclic mappings; trade-offs with communication costs

	onal			
Matrix computations	Optional			Mapping and load balancing
• Matrix product	Optional			Canon algorithm
• Linear systems	Optional			Data partitioning
• Matrix arithmetic	Optional			
• Matrix transpose	Optional			
• Termination detection	N	2	ParAlgo (>=K)	
• Leader election/symmetry breaking	N	2	ParAlgo (>=K)	

7.5 Cross Cutting and Advanced Topics

Table 4 Cross Cutting

Topics	BLOOM#	HOURS	Where Covered	Learning Outcome
High level themes:				
Why and what is parallel/distributed computing?	K	1/2	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.
Concurrency topics				know these underlying themes
Concurrency	K	1/2	CS2, DS/A	The degree of inherent parallelism in an algorithm, independent of how it is executed on a machine
Non-determinism	K	1/2	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	K	1/2	Systems, DS/A	Know that power consumption is a metric of growing importance, its impact on architectural evolution, and design of algorithms and software.
Locality	C	1	DS/A, Systems	Understand this as a dominant factor impacting performance - minimizing cache/memory access latency or inter-processor communication
Current/Hot/Advanced Topics				
Cluster	K	1/4	CS2, DS/A, System	Know as popular local-memory architecture with commodity compute nodes and an interconnection network
Cloud/grid	K	1/4	CS2, DS/A, System	know that both are shared distributed resources - cloud is distinguished by on-demand, virtualized, service-oriented software and hardware resources
P2P	k	1/4	CS1, CS2	Server and client roles of nodes with distributed data
Fault tolerance	K	1/2	Systems	Large-scale parallel/distributed hardware/software systems are prone

				to components failing but system as a whole needs to work
Security in Distributed System	K	1/2	Systems	Know that distributed systems are more vulnerable to privacy and security threats; distributed attacks modes; inherent tension between privacy and security
Distributed transactions	K	1/4	CS1,CS2, Systems	Know that consistency maintenance is a primary issue in such transactions involving concurrent updates from distributed locations
Web search	K	1/4	CS1, CS2	Know that popular search engines employ distributed processing for information gathering and employ distributed hardware to support efficient response to user searches
Social Networking/Context	N			Know that the rise of social networking provides new opportunities for enriching distributed computing with human & social context
Collaborative Computing	N			
Performance modeling	N			Know basic performance measures and relationships between them for both individual resources and systems of resources.
Web services	N			
Pervasive computing	N			
Mobile computing	N			

8. 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
By the target machine model			
SIMD	1	1	
Shared memory		1	1
Distributed memory	1	1	
Client Server		1	1
Hybrid	1		
By the control statements			
Task/thread spawning		1	1
SPMD		1	1
Data parallel		1	1
Parallel loop		1	
Parallel Programming notations			

Array languages			
Microprocessor vector extensions	1		
Fortran 90 and successors/C++ array extensions			
Shared memory notations			
language extensions		1	1
compiler directives/pragmas		1	
libraries		1	1
SPMD notations		1	1
CUDA/OpenCL			
MPI			
Global arrays			
BSP			
Functional/logical languages			
Parallel Haskell			
Parlog			
Semantics and correctness issues			
Tasks and threads		1	1
Fixed number of virtual processors			
Variable number of virtual processors.			
OpenMP implicit tasks			
Synchronization		1	1
Critical regions			
Producer-consumer			
Monitors			
Concurrency defects		1	
Deadlocks			
Unwanted Races			
Memory models			
Sequential consistency			
Relaxed consistency			
Tools to detect concurrency defects		1	
Performance issues			
Computation		1	1
Computation decomposition strategies			

Owner's compute rule			
Decomposition into atomic tasks and then coalescing ?			
Work stealing			
Elementary program transformations			
Loop fusion			
Loop fission			
Skewing			
Load balancing			
Scheduling and mapping			
Static			
Dynamic			
Data		1	
Organization of the data			
Data distribution			
Block			
Cyclic			
Block-cyclic			
Data layout			
Data locality			
False sharing			
Message aggregation			
Performance monitoring		1	
Tools			1
Performance metrics		1	
Speedup			
Efficiency			
Redundancy			
Isoefficency			
Amdahl's law			

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

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				

9. Appendix II: Suggestions on how to teach topics

9.1 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 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.). 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 tend 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. 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.

9.2 Programming

Parallel Programming paradigms:

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.

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.)

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

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

Parallel Programming notations:

Array languages:

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 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.

Language extensions:

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.

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.

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)

9.3 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 related to the input data size. (2) Adapt the notion in the parallel context to express the parallel time complexity as O(g(n,p)), where g is a function of n and the number of cores, 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 the notion of cost-optimality; i.e., a parallel algorithm is asymptotically cost optimal if p X 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 p X parallel space is of the same order as serial space).

speedup: Introduce the notion of speedup and give formal definition. Give a simple example, say adding n numbers in O(log n) time in parallel with p = n/2. Relate to cost optimality.

Scalability in algorithms and architectures: Revisit the adding n numbers example, show that speedup higher than O(n/log n) can be obtained when p << n. Use the example to show that speedup is a function of 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 and write conflicts; (ii) Illustrate its 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 pseudocodes; Highlight that the simple PRAM model empowers one to explore how much concurrency is available in a problem while not burdened with memory access and synchronization aspects.

BSP/CILK: Introduce BSP highlighting iterative computation wherein multiple processors compute independent subtasks followed by a global synchronization phase where processors communicate. The network latency 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.

Notions from scheduling: Take a simple problem such as max or sum of 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 of a reduction tree or of a centralized, hub-spoke tree, with all local sums updating a global sum). Use this to illustrate the task graph and the dependencies among parent and child tasks. Alternatively, consider floating point sum of two real values, and show its control parallel decomposition into a pipeline. Use this to illustrate the task graph and the 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 these task graphs modeling dependencies. Demonstrate scheduling among processors when there are fewer processors than amount of parallelism at a given level of task graph and processor reuse from level to level.

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

(make)span: Demonstrate identifying critical paths in a task graph and calculate a lower bound on parallel time (possibly using big-omega notation). Give examples (solving a triangular linear system, or Gaussian elimination).

Algorithmic Paradigms:

Divide & conquer (parallel aspects): Introduce simple serial algorithms, such as mergesort and/or Strassen's matrix multiply and then show how to obtain a simple parallel formulation 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: Highlight that this pattern is the natural way to solve a problem, which needs more than one phase/sub-algorithm due to data dependency to solve a problem. Illustrate it using 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). May show the task graph and identify the critical path as the composition of individual critical paths of the constituent phases.

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 det.:
- 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

9.4 Crosscutting

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: can be covered briefly and then highlighted in various contexts

Non-determinism: e.g., floating point arithmetic in parallel can lead to different results; example of web search on various sites can lead to better results or parallel branch and bound usually finds better solution because of non-determinacy.

Power: Discuss importance of energy efficiency from multiple perspectives, e.g., unsustainable power/thermal densities, long lifetime needs of embedded & mobile devices, energy costs, and sustainability considerations. Optional: Discuss metrics & tradeoffs between power, energy, thermal vs. performance & reliability; a brief discussion of sustainability issues (e.g., e-waste).

Locality: Its manifestation in cache vs. main memory, memory vs. hard drive, local memory vs. non-local memory, 1-hop neighbor vs. distant neighbor; Can be illustrated examples of two algorithms of same computational complexity for solving a problem... one with local data-access pattern and one with a non-local access pattern. Students implement the two seemingly simple variants of the same algorithm and observe the difference in performance. A very simple example would be dense matrix multiplication with three nested loops. Simple variations in organizing the loops typically result in a wide fluctuation in performance. Highlight in several other context.

Current/Hot/Advanced Topics:

Cluster: e.g., Beowulf cluster

Cloud/Grid: Example: Google Doc - collaborative computing (editing), service in the cloud, and seti@home for volunteer grid computing

P2P: file/music-sharing systems, multi-party games using Bluetooth devices (such as Nintendo DS/2). Comparison between client-server and P2P approaches to problem solving. Optional: Structured vs. unstructured P2P. Overview of bit-torrent & other popular P2P SW.

Fault Tolerance: e.g. techniques for fault tolerance: redundancy, check pointing for software, types of faults (byzantine, fail-stop, etc.); redundant paths to route message on the Internet (or in a parallel system); Optional: Reliability in semiconductor processes

Security in Distributed System: e.g. buffer overflows can be exploited for security attacks; distributed denial of service attacks, exploiting mutually inconsistent configuration of sub-systems and multiplicity of components;

Distributed transactions: Examples: ATM transactions (strong consistency); Social Networking sites (weak consistency): giant distributed databases; rich environment for illustrating distributed security and distributed transactions

Web search: e.g. web crawling and its cluster architecture

Social Networking/Context: Discuss how social networking can be (and already is being) exploited to enrich e-commerce and enhance collaboration services but also presents challenges in the areas of security, privacy, efficient data dissemination, and context determination.

Performance Modeling: E.g., utilization, backlog (queue length), response time, Little's law, stability, closed systems (degree of multiprogramming, thread stalls), elementary bottleneck analysis, memoryless & PASTA property, work conservation, etc. Could illustrate by showing how doubling of CPU speed may speed up the application only by a few percent.

Web Services: Talk about web-services paradigm & ingredients (e.g., UML/HTML, HTTP(S), SOAP, WSDL, UDDI, …) and to put together simple web-services

10. 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)