NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates Version 2.0_{β}[1][2]

November 2020

Website: https://tcpp.cs.gsu.edu/curriculum/?q=home

Curriculum Working Group:

Prasad, Sushil K. (Coordinator; University of Texas at San Antonio),
Estrada, Trilce (Big Data Aspect Lead; University of New Mexico),
Ghafoor, Sheikh (Cross-cutting Aspect Lead; Tennessee Tech),
Gupta, Anshul (Algorithms Area Lead; IBM T. J. Watson Research Center),
Kant, Krishna (Energy Aspect Co-lead; Temple University),
Stunkel, Craig (Energy Aspect Co-lead; IBM T. J. Watson Research Center),
Sussman, Alan (Programing Area Lead; University of Maryland, NSF),
Vaidyanathan, Ramachandran (Distributed Aspect Lead; Louisiana State University),
Weems, Charles (Architecture Area Lead; University of Massachusetts),
Agrawal, Kunal (Washington University, St. Louis),
Barnas, Martina (Indiana University, Bloomington),
Brown, David W. (Elmhurst University),
Bryant, Randy (Carnegie Mellon University),
Bunde, David P. (Knox College),
Busch, Costas (Louisiana State University),
Deb, Debzani (Winston-Salem State University),
Freudenthal, Eric (University of Texas, El Paso),
Jaja, Joseph, (University of Maryland),
Parashar, Manish (Rutgers University, NSF),
Phillips, Cynthia (Sandia National Laboratory),
Robey, Bob (Los Alamos National Lab),
Rosenberg, Arnold (Northeastern University),
Saule, Erik (University of North Carolina, Charlotte),
Shen, Chi (Kentucky State University),

[1] Version 1 was released in Dec 2012 (http://tcpp.cs.gsu.edu/curriculum/?q=system/files/NSF-TCPP-curriculum-version1.pdf). This is a draft for community input in preparation for the release of Version 2. Contact: Sushil K. Prasad, sushil.prasad@gmail.com

[2] How to cite this report: Prasad, S. K., Estrada, T., Ghafoor, S., Gupta, A., Kant, K., Stunkel, C., Sussman, A., Vaidyanathan, R., Weems, C., Agrawal, K., Barnas, M., Brown, D. W., Bryant, R., Bunde, D. P., Busch, C., Deb, D., Freudenthal, E., Jaja, J., Parashar, M., Phillips, C., Robey, B., Rosenberg, A., Saule, E., Shen, C. 2020. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version II-beta, Online: http://tcpp.cs.gsu.edu/curriculum/, 53 pages.

Abstract:

In this era of big data, cloud, and multi- and many-cores, the computer science (CS) and computer engineering (CE) graduates must have basic skills in parallel and distributed computing (PDC). This curriculum guideline includes a core set of PDC topics that a student graduating with a Bachelor's degree in CS or CE would be well-served to have covered through required courses and is intended for use by instructors and their students, employers, authors and publishers, curriculum committees, professional societies and other stakeholders. It also contains additional topics suitable for incorporation into advanced or elective courses. The topics are primarily organized into the areas of architecture, programming, and algorithms topics. A set of pervasive concepts that percolate across area boundaries are also identified, and most core topics in the three areas support building students' knowledge and comprehension of the pervasive concepts and their various manifestations. Version 1 of this curriculum was released in December 2012. That curriculum guideline has over 100 early adopter institutions worldwide and informed the 2013 ACM/IEEE Computer Science curricula. This report is a major revision of those guidelines. The updates have focused on enhancing coverage related to the topical aspects of Big Data, Energy, and Distributed Computing. Topics from these aspects have been integrated into the three area tables and a fourth table for emerging topics, a few previous topics have been eliminated, and the four tables have been reorganized. This report contains an introductory write up on curriculum's need and rationale, background, companion, and future activities, followed by the proposed topics, each with an expected Bloom level of coverage and learning outcomes for core courses and where appropriate a Bloom level and learning outcomes for advanced courses, and suggestions for relevant courses. A companion activity has begun to collect and create exemplars for topics and courses to aid instructors, which will enrich the material going forward and will be cross-referenced against the topics. This report is expected to engage various stakeholders for their feedback and adoption, and for participation in ongoing activities.

Table of Contents

1. Introduction

1.1 Motivation

Parallel and Distributed Computing (PDC) now permeates most computing activities. Multiple cores and general-purpose graphics processing units (GPUs) are common even on laptops and handheld devices, and many productivity tools depend on distributed services. PDC is not just an integral part of the work of computing professionals who *explicitly* design systems that exploit concurrency to achieve performance; it is also relevant to developers and users of applications that hide much of the complexity of harnessing PDC technology. These *implicit* consumers of PDC may include developers with applications that interface with everyday tools or libraries such as collaborative environments, productivity tools, and multimedia applications that utilize local and/or remote PDC technology implemented below their visibility threshold. The penetration of PDC into the daily lives of both "explicit" and "implicit" users has made it imperative that all computing professionals be able to understand its scope, effectiveness, efficiency, and reliability.

The preceding trends point to the need for imparting a broad-based skill set in PDC technology at various levels in the educational fabric of Computer Science (CS) and Computer Engineering (CE) programs as well as related computational disciplines. Thus, it is imperative that all computing professionals, including those whose interaction with PDC is only implicit, are familiar with its basic concepts and the interactions and implications of these concepts upon the semantics of systems they exploit.

The Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER), in conjunction with the IEEE Technical Committee on Parallel Processing (TCPP), developed undergraduate TCPP curriculum guidelines for PDC during 2010-12[3], the first version of this curriculum, with the premise that *every* computing undergraduate should achieve a specified skill level regarding PDC-related topics as a result of *required* coursework. The TCPP curriculum informed the ACM/IEEE 2013 Computer Science Curricula[4] and has been broadly adopted. To keep up with the dynamic landscape of computing research and practice, the 2012 TCPP curriculum has now been revised, especially incorporating aspects of Big Data, Energy, and Distributed Computing.

1.2 Our Vision and Intended Use

We recognize that any revision of a core curriculum is a long-term community effort. Our vision has been one of stakeholder experts working together, and periodically providing guidance on restructuring standard curricula across various courses to incorporate parallel and distributed computing. The primary beneficiaries are CS/CE students and their instructors who receive up to date guidelines identifying aspects of PDC that are important to cover and the suggested specific courses in which their coverage might find an appropriate context. Various programs at colleges, nationally and internationally, can receive guidance in setting up courses and/or integrating parallelism within the Computer Science, Computer

[3] NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates. http://tcpp.cs.gsu.edu/curriculum/?q=system/files/NSF-TCPP-curriculum-version1.pdf
[4] ACM/IEEE-CS 2013 Computer Science Curricula. https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

Engineering, or Computational Science curriculum. Employers can have a better sense of what they can expect from students in the area of parallel and distributed computing skills. Curriculum guidelines can similarly help inform retraining and certification for existing professionals as well as prepare the ground for curriculum setting professional societies for their periodic updates.

Our larger vision in proposing and updating this curriculum has been to enable students to be fully prepared for their future careers in light of technological shifts and mass adoption of PDC through multicores, GPUs, cloud computing, big data, IoT and corresponding software environments, and to make a real impact with respect to all stakeholders, including employers, authors, and educators. These curricular guidelines, along with periodic feedback and other evaluation data on their adoption and use, will also help to steer companies hiring students and interns, hardware and software vendors, and, of course, authors, instructors, and researchers.

This updated curriculum proposal continues to seek adoption and use in a manner that is flexible and broad, always allowing for local variations in emphasis. The field of PDC is changing too rapidly for a proposal with any rigidity to remain valuable to the community for a useful length of time. However, it is essential that curricula begin the process of incorporating parallel thinking into the core courses. Therefore, this curriculum attempts to identify basic concepts and learning goals that are likely to retain their relevance for the foreseeable future. We see PDC topics as being most appropriately sprinkled throughout a CS/CE curriculum in a way that enhances what is already taught and that melds parallel and distributed computing with existing material in whatever ways are most natural for a given institution/program. The document also includes additional elective topics for upper level courses. While advocating the thesis that relegating the most basic PDC subjects to a separate course is not the best means to shift the mindset of students away from purely sequential thinking, we recognize that the separate-course route may work better for some programs, particularly for the coverage of advanced/elective topics.

1.3 The Curriculum and its Update

Architecture, Programing and Algorithms Areas: For the three main PDC *areas* of Architecture, Programming, and Algorithms, starting from the previous area tables, the working group reconsidered various topics and subtopics and their level of coverage, identified in which current lower level core courses these could be introduced, and provided examples of how they might be taught. For those topics which are either not suitable for coverage in lower level core courses or for which deeper treatment can benefit an undergraduate student, their level of coverage and learning outcomes for advanced or elective courses have also been identified. For each topic/subtopic, the process involved the following.

1. Assign a learning level using Bloom's classification[5] using the following notation.
   K= Know the term
   C = Comprehend so as to paraphrase/illustrate
   A = Apply it in some way
2. Write learning outcomes and teaching suggestions for core courses, and if warranted, for advanced/elective courses.
3. Identify core and advanced/elective courses where the topic could be covered.

[5] (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.

A fourth table for *emerging topics*, which could not be accommodated within the three areas, have also been identified and updated. These are of significant current and emerging interests, and are still evolving. These topics are in general better suited for upper division classes but may be introduced in the early core courses in a limited way. A few previous topics have been eliminated from these four tables and the tables have been reorganized.

Pervasive Concepts: A set of pervasive concepts that percolate across area boundaries are also identified, and most core topics in the three areas support building students' knowledge and comprehension of the pervasive concepts and their various manifestations. It is desirable for educational programs to enable students to comprehend PDC's pervasive concepts in multiple contexts. Four key concepts that impact the performance, correctness, and semantics of PDC systems in a pervasive manner include the following (while recognizing that this list is a work in progress):

- Concurrency: The availability and exploitation of simultaneous or overlapping actions.
- Asynchrony: How concurrency can enable actions to occur in multiple orderings, and how this affects the design of programs that achieve high performance and correctness.
- Locality: That data or computational subsystems are local only to the system that actually contains it, remote access imposes delays, costs, and potential inconsistencies. Locality is relevant at multiple levels, and high performance and correctness requires that it be effectively managed.
- Performance: Metrics for characterizing throughput, speedup, efficiency, scalability, etc., at various levels of abstraction.

Big Data, Energy, and Distributed Computing: The majority of updates to the curriculum have focused on enhancing coverage related to the topical *aspects* of Big Data, Energy, and Distributed Computing. Topics from these aspects have been integrated into the three area tables and the table for emerging topics. The Big Data aspect was introduced in response to the increasing need for PDC in data intensive problems, the growing demand for a skilled workforce in data science and machine learning, and the rise of academic programs in data science, data analytics[6] and machine learning. While other guidelines[7] [8] [9] addressing data science in undergraduate programs focus on computational and statistical thinking in general, our goal is to address data science challenges in the PDC context. Topics of relevance include hardware and software support for data collection, storage, organization, and processing; constraints imposed by I/O and memory hierarchies; performance bottlenecks due to data movement; and parallel algorithmic approaches useful for massive data analyses.

[6] 50 Best Big Data Degrees, collegechoice.net, https://www.collegechoice.net/rankings/best-big-data-degrees/

[7] A. Danyluk, P. Leidig, L. Cassel, and C. Servin. 2019. ACM Task Force on Data Science Education: Draft Report and Opportunity for Feedback. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). ACM, New York, NY, USA, 496-497

[8] Curriculum Guidelines for Undergraduate Programs in Data Science, PCMI 2016 https://www.stat.berkeley.edu/~nolan/Papers/Data.Science.Guidelines.16.9.25.pdf

[9] Curriculum Guidelines for Undergraduate Programs in Statistical Science (ASA), [ASA 2015] http://www.amstat.org/education/pdfs/guidelines2014-11-15.pdf

Energy and power have emerged as key concerns in computing in the last 15 years or so, and are usually not parts of traditional coverage in undergraduate CS or CE curricula. However, with power consumption issues becoming the primary roadblock to increasing single thread performance, and a key stimulus for increased parallelism and heterogeneity, it is imperative that it is discussed in the undergraduate PDC curriculum. Topics of relevance include the basics of dynamic and static power management for individual cores and sockets, and how related techniques can be exploited in the PDC context.

Traditionally, elements of distributed computing (such as mutual exclusion and synchronization) have been discussed in undergraduate curricula. However, the proliferation of independent and untethered computing devices and applications such as collaborative environments and cloud services has made the early discussion of distributed computing concepts vital. These new applications also present an opportunity to introduce PDC ideas against a backdrop that most students would readily identify with. The revised curriculum has distilled some of the core ideas of distributed computing, often weaving them through several conventional settings before fully exploring them in advanced courses on distributed systems or parallel programming.

Organization: This rest of the report is organized as follows. Section 2 describes the background work on developing the PDC curriculum, its impact so far, and companion activities and future work. Section 3 then explains how to read the curriculum in a manner consistent with its underlying intent. Sections 4 describes the Pervasive concepts and their rationale. Sections 5, 6, and 7, respectively, provide rationale and tables for each of the major topic areas in the curriculum: architecture, programming, and algorithms. Finally, Section 8 provides the table for the emerging topics.

2. Background and Activities

We start with some historical background behind the NSF/TCPP curriculum effort and a synopsis of some synergistic activities of the curriculum working group. Readers who wish to refer to the curriculum guidelines only may skip to Section 3.

2.1 Version 1 of NSF/TCPP Curriculum and Impact

As background preparation for the development of the curriculum proposal, a planning workshop funded by the US National Science Foundation (NSF) was held in February 2010 in Washington, DC. This was followed up by a second workshop in Atlanta, alongside the International Parallel and Distributed Processing Symposium (IPDPS) 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 term. The planning workshops and their related activities benefited from experts and various stakeholders, including instructors, authors, industry, professional societies, NSF, and the ACM education council. The primary task identified was to propose a set of core topics in parallel and distributed computing for undergraduate curricula for CS and CE students. Further, it was recognized that, in order to make a timely impact, a sustained effort was warranted. Therefore, a series of weekly/biweekly tele-meetings was begun in May 2010. These weekly tele-meetings have continued uninterrupted since then. A preliminary version of the NSF/TCPP curriculum guidelines was released in Dec of 2010 and the first version was released in 2012, after nearly two years of deliberations. The first version of the curriculum can be found at http://tcpp.cs.gsu.edu/curriculum/?q=system/files/NSF-TCPP-curriculum-version1.pdf.

2.2 Impact of the Curriculum

This initial effort was quite impactful. The CS2013 ACM/IEEE Computer Science Curriculum Joint Task Force also recognized the need to integrate PDC topics in the early core courses in the CS curriculum, and collaborated with us leveraging our curriculum. The CS2013 curriculum explicitly refers to the NSF/TCPP curriculum for comprehensive coverage of parallelism (and provides a direct hyperlink[10]). ABET/CSAB now has a new parallelism requirement[11], and we have initiated collaborations. The enthusiastic reception of the curriculum guidelines led to a commitment within the working group to continue to develop the guidelines and to foster their adoption at an even broader range of academic institutions. Toward these ends, the NSF-supported Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER) was founded[12]. Since 2011, CDER has worked for broad adoption of PDC in early courses by awarding over 100 early adopter grants, organizing four annual Edu* workshops and five CyberTraining workshops, and publishing two edited books and two journal special issues.

Early Adopter Competitions: A vibrant community of early adopters and educators has taken hold, who are carrying the torch to various parts of the world, enriching the discourse, and contributing resources all the while shining light on emerging needs and further work, including the need to update the curriculum. Since the release of our curriculum guideline, we have run early adopter competitions six times: in spring and fall 2011, spring and fall 2012, and fall 2013, 2014, and 2015, selecting a total of 141 institutions. NSF and Intel supported the grants, with hardware donations from NVIDIA. Aspiring early adopters submitted a proposal that was evaluated by a CDER committee and external experts. Selected proposals were awarded a small stipend, ranging from $1K for single course proposals to $2.5K for multi-course multi-semester proposals. International awards included India, China, Turkey, Spain, Argentina, Malaysia, and others.

In order to allow the early adopters, the public, and the working group to benefit from everyone's experiences and evaluations, the first ***EduPar*** workshop, collocated with IPDPS in Anchorage, was organized in May of 2011. Thereafter, EduPar has been a regular annual workshop at IPDPS. We also initiated a new workshop on education at SC13 in Denver, ***EduHPC***. This workshop was specifically dedicated to bringing together stakeholders from industry (hardware and software vendors and employers), government labs, funding agencies, and academia, so that each could hear the challenges faced by the others, learn the various approaches to these challenges, and generally have opportunities to exchange ideas and brainstorm solutions. This again has become an annual workshop at SC. Working with colleagues in Europe, we also helped launch a new workshop in conjunction with EuroPar 15 in Vienna, called ***Euro-EduPar***. The recent addition to Edu* series has been EduHiPC at HiPC'18.

[10] The ACM/IEEE Computer Science Curricula 2013: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf
[11] ABET Criteria for Accrediting Computer Science and Similarly Named Programs, https://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2019-2020/#GC5
[12] NSF-supported Center for Curriculum Development and Educational Resources (CDER): https://tcpp.cs.gsu.edu/curriculum/?q=node/21183

CDER has additionally initiated several complementary activities toward the goal of fostering PDC education.

- A *courseware repository*[13] has been established for pedagogical materials (exemplars, sample lectures, recommended problem sets, peachy assignments, experiential anecdotes, evaluations, papers, etc.). This is a living repository. CDER invites the community to contribute existing and new material to it. The Exemplars group is working to provide an extensive set of exemplars for various topics and courses.

- A *CDER Compute Cluster*[14] has been set up for free access by early adopters and other educators and their students. The CDER cluster was upgraded in Fall'18 with additional GPUs and the Spark environment. Its current specifications are 656 cores, 1 TB of RAM, and four GPUs, including NVIDIA V100s, and includes an Apache Spark subsystem.

- A *book project* was initiated recognizing that both instructors and students need suitable textual material. The book project's goal is to address the lack of suitable textbooks to integrate PDC topics into the lower level core courses (CS1, CS2, Systems, Data Structures and Algorithms, Logic Design, etc.). The first edited book volume entitled "Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses" was published September 2015 in hardcopy by Elsevier with a free preprint version on the Web[15]. This volume was focused on the introductory courses, and the preprint version has had over 39,000 chapter downloads. A second book was published by Springer in Fall 2018, entitled "Topics in Parallel and Distributed Computing: Enhancing the undergraduate curriculum - performance, concurrency, and programming on modern platforms." It provides exemplars that are geared more toward upper level electives. Both books have two categories of chapters, one meant primarily for instructors, and the other for students.

2.3 Companion Curriculum Activities

A companion activity to curriculum development has also begun with the ambitious goals of collecting, curating and developing exemplars of various kinds: (i) *Topic exemplars* on how to teach individual or related PDC topics and suitable instructional materials; (ii) *course exemplars* on how entire core CS and CE courses can be constituted to infuse suitable PDC topics from the TCPP curriculum; and (iii) *curriculum exemplars* in adapting the entire CS/CE curriculum for alignment with TCPP curriculum guidelines.

A second companion activity that NSF has recently funded the CDER group for is to explore a more Computer Engineering-oriented version of the TCPP curriculum, primarily focusing on PDC topics which can be infused into early CE core courses. CS and CE programs have similar early core courses, such as introductory programming, discrete math, and data structures and algorithms. However, CE offerings also differ from CS, for example, with emphasis on topics such as digital logic, and computer communications and interfacing. In addition to collaboration with the ACM/IEEE taskforce for their CS curricula, this component of our work could inform the taskforce for their CE curricula in their next revision cycle.

This report is expected to engage various stakeholders for their feedback and adoption as well as for participation in ongoing activities.

[13] CDER Courseware Repository: https://tcpp.cs.gsu.edu/curriculum/?q=courseware_management
[14] CDER Cluster free access: https://tcpp.cs.gsu.edu/curriculum/?q=node/21615
[15] CDER Book Series - Free preprint version on the Web at https://tcpp.cs.gsu.edu/curriculum/?q=CDER_Book_Project

3. How to use these Guidelines

The premise behind the guidelines in this document is that *every* CS/CE undergraduate should achieve a certain minimum skill level in the basic 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 may be 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. Recognizing this, we went beyond the core curriculum and also identified a number of topics that we recommend for inclusion in advanced and/or elective curricular offerings; however, we do not insist that every CS/CE undergraduate student acquire a certain level of literacy in each of these higher-level topics.

Through the feedback on the first version of the curriculum guidelines, we recognized that whenever it is proposed that new topics be included in the curriculum, often the readers automatically assume that something else will need to be eliminated. However, for many of the topics we propose, this is not the case; rather, it is more a matter of changing the approach of teaching traditional topics to encompass the opportunities for "thinking in parallel." For example, when teaching array-search algorithms, it is quite easy to point to places where independent operations could take place in parallel, so that the student's concept of search is opened to that possibility. In some cases, we are indeed proposing material that may require making choices about what it will replace in existing courses. Since we only suggest *potential* places in a curriculum where topics can be added, we leave it to individual departments and instructors to decide whether and how coverage of parallelism may displace something else.

The reader should keep in mind that many of the topics discussed in this proposal can appear at multiple levels in the curriculum. Upon seeing a topic, the reader should try not to make a premature judgment regarding its suitability for an undergraduate course. Rather, the reader should consider where and how aspects of the topic might naturally be blended into a suitable context to facilitate the move to holistically allowing students to develop a capacity for parallel and distributed thinking. Further, the suggested level of proficiency for a topic may be achieved though coverage in progressively increasing depth, spread over several modules in one or more courses.

For each of the topics, we suggest where and, in many cases, how it can be covered in a curriculum. These are suggestions and examples, rather than prescriptions, with the goal of illustrating possibilities and encouraging the reader to think about multiple approaches. The curriculum guidelines are not meant to specify precisely where each topic is addressed. Our intention is, rather, to encourage instructors to find as many ways as appropriate to insert coverage of the indicated PDC topics into core courses. Even comments of a few sentences about how a topic under discussion can be seen from a new perspective in a parallel or distributed context, when judiciously sprinkled throughout a course, will help students to expand their PDC thinking. Students can start to think in parallel and distributed terms when they sense that their instructors are always conscious of the implications of parallelism and distributed computing with respect to each topic that is covered in their courses, including topics in which the parallel or distributed content may not be obvious.

In summary, we recommend that instructors make every attempt to incorporate PDC ideas in existing courses, rather than change the courses drastically. The recommended topics can be covered up to the recommended level of depth in many ways and across multiple combinations of courses; our course recommendations are simply our best guesses of the courses where certain topics may fit in a typical curriculum. The learning outcomes are meant to be satisfied at the end of the curriculum and not necessarily at the end of any particular course.

We summarize the notations and conventions used in the curriculum guidelines that follow in the subsequent sections.

3.1 Notations and Conventions

Each table enumerates the broad categories of concepts in Sections 4-8, and breaks these down to narrower ideas and topics, some to multiple levels of detail. In addition, a topic also contains a Bloom Level to indicate a level of coverage for the topic, a (set of) course(s) in which the topics could be covered, and learning outcomes and pointers to achieving them. We briefly explain each of these considerations below.

Concept Format: The hierarchy of concepts is indicated in the tables as follows:

- Top level topic groups are shown in bold.
- Second level topic groups are shown in italics.
- Third level topics, which sometimes have subtopics, are shown in regular text.
- Fourth level sub-topics are shown with bullets.

This hierarchy, whose depth varies across topics/concepts, reflects the need to elaborate on certain ideas more than others. In most cases, details (Courses, Bloom levels, etc.) are provided for the lowest level of the hierarchy. In some cases, higher levels of the hierarchy also provide some of this information, where appropriate.

Courses: The courses to which topics are mapped are broadly divided into “Core Courses” and “Advanced Courses.” This separation and the listing of courses below is not a rigid partition, rather it is a rough separation of ideas into commonly used course names. With the recognition that the names of the course and their coverage vary widely across institutions, the reader should interpret the “courses” in a manner that is most useful for his/her particular context. In this setting, courses are listed in order of likelihood of where the topic can be included. Once again, it should be noted that coverage of an individual topic/concept may span multiple modules in a course, or multiple courses.

- Core (lower level) courses include:

CS1	Introduction to Computer Programming
CS2	Second Programming Course in the Introductory Sequence
DS/A	Data Structures and Algorithms
Systems	Introductory Systems/Architecture Course

- Advanced/elective courses include:

Algo2	Elective/Advanced Algorithm Design and Analysis

Arch2	Advanced/Elective Course on Architecture
Compilers	Compiler Design
DB	Database Systems
DistSystems	Distributed Systems
Lang	Programming Languages/Principles
Netw	Communication Networks
OS	Operating Systems
ParAlgo	Parallel Algorithms
ParProg	Parallel Programming
SwEng	Software Engineering

Bloom Levels: The level of coverage and expected learning of each topic is expressed using Bloom's classification[16] with the following notation[17]:

K= Know the term (basic literacy)
C = Comprehend so as to paraphrase/illustrate
A = Apply it in some way (requires operational command), analyze the concept and be able to use it in other settings.

The indicator "N" denotes a topic whose coverage is best deferred to a non-core advanced course.

Learning Outcomes and Teaching Suggestions: These indicate a rough expectation of student understanding of the topic at the end of the curriculum. In many cases, this column of the tables often also includes suggestions to the instructor of possible ways to explore the topic. To provide better clarity and context, the learning outcomes themselves are separated for core and non-core courses.

4. Pervasive PDC Concepts

There are a few broad themes that are conceptual underpinnings of parallel and distributed computing. These themes are fundamental in nature, cut across all areas of PDC (algorithms, architecture, and programming), and in some cases, may transcend PDC itself. They appear repeatedly at different depths and complexity throughout the curriculum. Many of the PDC topics are related to and stem out of these pervasive themes. Some of these concepts are inter-related to each other. Because of their pervasive nature, it is recommended that coverage of these concepts be made an explicit and essential part of a PDC curriculum. These themes should be introduced very early in the curriculum, often through unplugged activities. It is also recommended that these pervasive topics be further reinforced in later classes in multiple context, depth, and complexity as appropriate in

[16] (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/cognition/bloom.html
[17] Some advanced topics are identified as "N" as being "not in core" but which may be included in an elective course.

the architecture, programming, and algorithm areas. At the basic level, these concepts should be taught at least at "C" level in terms of Bloom classification. While we have identified four such pervasive topics, we recognize that this list is a work in progress.

Concurrency**:** Algorithms are generally specified as sequential programs in languages that explicitly specify a set of variables and a sequence of operations to be performed upon them. In order to reduce the time required to execute an algorithm through parallel speedup, its sequential specification can be relaxed to permit individual operations or their groups to be executed concurrently by multiple operational units (e.g., thread, CPU). Hence, concurrency is a property of an algorithm, it exposes potential for parallelization. If concurrency is present in an algorithm then the concurrent operations can be executed in parallel (simultaneously) by multiple operation units (CPU's) if available, without concurrency there is no scope for parallelization. Concurrency can be present in a sequential program, parallelization takes advantage of concurrency to increase performance. When there is a sequential dependency among operations, mechanisms must be provided to forward data among operational units. When the result of one early operation affects the sequence of future operations, the speculative early execution of multiple potential future computational paths may yield additional speedup. This decomposition of complex tasks in a manner that exploits and manages available parallelism at multiple scales occurs in other contexts that are accessible to students. For example, a kitchen staff in a restaurant will concurrently prepare multiple meals, and may speculatively begin to prepare some dishes prior to the arrival of customer orders. Optimization of such a kitchen's operations involves nuanced analysis of available concurrency, speculation, and coordination needs such as using warming stations to enable concurrency.

Asynchrony**:** Both Turing machines and beginning programmers assume that when an operation is invoked, it is completed immediately. Modern computers, however, are increasingly asynchronous for reasons of performance and efficiency. This characteristic is similar across CPU cache lines, I/O buffers, networks, and database entries. Asynchrony is primarily locality in time where operations are not instantaneous, but it also can manifest itself as locality in space (e.g. cache and main memory have different data values). The observance of this phenomena can be introduced in the entry-level Computer Science classes (e.g. all students open up a shared google docs spreadsheet and everyone enters their name in cell A1). This sets the stage for discussions on how to design mitigations in later courses. It is also important to tie the generality of the concept across scales of architecture from CPUs to distributed networks as manifestations of the same basic concept — that operations cannot truly be instantaneous and that clocks are not perfectly synchronized. In short, in computer science, time is messy. Of course, this issue of time goes beyond Computer Science. Clocks became widespread due to the introduction of trains and the increasing accuracy of clocks was driven by ocean navigation. Races become possible due to asynchrony. Much of the complexity of modern programming is trying to impose the perception of synchronous behavior where it doesn't really exist. The difficulty of this task can be ascertained by the continuing issue of thread-safe libraries decades after threading was introduced in computers. But the issues of asynchrony extend far beyond just threads.

Locality**:** One of the overarching concepts in computing is that of locality of both time and space. Like concurrency, locality is a concept that predates computer science. Armies are recursively divided into regiments and groups of individual soldiers that coordinate at various levels of granularity. Caches of supplies such as ammunition must be actively managed and coordinated. Many animals benefit from social pack structures that productively exploit parallelism that includes some level of local autonomy within a larger organization. Computer architecture is similarly simplified through localized control. Each core executes its own instruction stream through prefetching and concurrently scheduling of independent operations onto multiple (pipelined) functional units, and those functional units (mostly) manage their internal resources. Each computational unit (a CPU in a shared memory machine or a node in a cluster) may have their own clock and their own notion of time. Memory subsystems proactively predict and cache future memory references based upon recent memory reference patterns. A challenge with localized control is the detection and management of conflict. For example, in order to provide the illusion of a single shared low-latency memory, each core of a CPU is provisioned with an independent local cache that implements a coherence protocol with the other caches. Effective memory access times are highly dependent upon access patterns: A program whose memory access patterns trigger frequent transfers of data among caches will consume more energy and have much higher access latency and lower memory throughput than programs which do not. There is tension between global and local control in the management of sub-tasks at every scale. Broad-scale coordination can require substantial allocation of resources for monitoring and micro-management that might otherwise be used towards speeding up computation, and the cost or delays required to compute and deploy optimized allocations may be onerous. Localized control trades global knowledge for simplicity and speed. However, myopic localized control can miss opportunities for global optimizations such as resource reallocation in response to asynchrony or localized faults.

***Performance*:** Another overarching concept that spans over all areas is performance. No matter what computing artifact (program algorithms, hardware) that are being designed, studied, and analyzed, one should be aware of how good the artifact is and strive to make it better. Students should be introduced to what to measure, how to measure, and what metric to be used to measure the goodness of an artifact . Space, time, and energy are the basic commodities to measure and the metrics for these commodities may differ based on whether the context is sequential or parallel. For example in a sequential program run time may be used as a measure of goodness but in the parallel version of the same program there is an additional variable, the number of cores, so the notion of runtime does not capture the goodness. Different metrics are needed for parallel and distributed programs (e.g. speedup, efficiency, etc.). Similarly asymptotic notations such as big O are used for sequential algorithms but for parallel algorithms scalability is a more appropriate metric for goodness.

Table 1. Pervasive Concepts

Pervasive Concepts			
Concepts	**Core BLOOM #**	**Where Covered**	**Learning Outcome**
Asynchrony	C	CS1; CS2; DS/A; A(Arch2, OS)	1) Understand cause and effect of Asynchrony and how to ensure that computational correctness. 2)Understand asynchrony is the characteristics of modern systems and understand asynchrony in PDC context. 3)1) Can utilize a standard coordination strategy to prevent incorrect operation due to uncontrolled concurrency (race conditions).
Concurrency and Dependency	A	CS1; CS2; DS/A; System	1) Understand concurrency is an algorithmic property and difference between concurrency and parallelism. 2) Understand sequential dependency limit degree of concurrency and hence parallelism. 3)Identify sequential dependency in algorithms.
Locality	A	CS1; CS2; DS/A; Systems	1)Understand the locality of space and time. 2) Can describe one case of locality affecting behavior(e.g. web cache). 3) Know about some algorithm/program sensitive to some sort of artifact of architectural locality. 4) Implement some programs that are optimized around some locality. 5) Aware of at least two forms of locality.
Performance	A	CS1; CS2; DS/A; Systems	1) Understand what to measure and how. 2) Understand space, time, & energy are fundamental commodities to measure. 3)Able to implement/tune some algorithms around performance metric. 4) Comprehend, explore, and analyze some speedup, efficiency, and scalability.

5. Architecture Topics

5.1 Rationale

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

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

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

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

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

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

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

The guideline divides architecture topics into six major areas: Classes of parallelism, underlying mechanisms, floating-point representation, performance metrics, power, and scaling. 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.

Classes of Parallelism topics are meant to encourage coverage of the major ways in which computation can be carried out in parallel by hardware. Understanding the differences is key to appreciating why different algorithmic and programming paradigms are needed to effectively use different parallel architectures. The classes include data and control parallelism, pipelining, and communication via shared memory or message passing.

Underlying Mechanisms has replaced the *Memory Hierarchy* section in this version of the curriculum because many of the associated concepts scale to distributed systems. Caching is covered in the traditional curriculum, but when parallelism and distribution come into play, the issues of atomicity, consistency, and coherence affect the programming paradigm, where they appear, for example, in the explanation of thread safe libraries. The revised curriculum also addresses the problem of false sharing and issues that arise as systems scale up to handle larger data sets. Additionally, interrupts and event handling are fundamental in many contexts involving concurrency and distribution, starting even from a basic understanding of graphical user interfaces. Handshaking as an alternative to synchronous communication is an aspect of communication between processors and devices at multiple levels of granularity. Lastly, we have added a topic on process and system ID as a basic necessity for identifying the source and destination in a communication, as well as symmetry breaking in programming parallel and distributed systems.

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.

Power has been added as a section in the revised curriculum because it has grown in significance since the original recommendations were issued. At the low end, mobile devices are very sensitive to power efficiency, and that has resulted in the use of multiple cores, sometimes with heterogeneous performance levels. At the high end, supercomputer architectures are constrained by the availability and cost of power. Most of this material is already being addressed in computer architecture courses.

Scaling has been added in the revision to reflect the realities of HPC and big data applications, where previously minor factors grow in significance to have major effects. These are mostly topics for upper level courses with some preliminary coverage in the core courses so that students are aware that the simple approaches they are initially learning are not going to be suitable for large problems. These topics include reliability, fault tolerance, data bound computation, memory bound computation, scale-out implications, data movement cost versus computation cost.

Many of the architecture learning goals are listed as belonging to an architecture course, or a systems course (which may be a traditional computer organization course, or a more general systems principles course). The teaching examples, however, describe ways that some can be introduced in lower level courses. Some topics are indicated as belonging to other advanced courses. These are not included in the core curriculum. We have included them as guidance for topical coverage in electives, should a department offer such courses.

5.2 Updates from version 1.0

The updates to the Architecture table are described in detail above. To summarize, the Memory Hierarchy section has been generalized and renamed Underlying Mechanisms with several new topics. Some less relevant benchmarking topics (such as means) were deleted from Performance Metrics. New sections on Power and Scaling have been added. Some updates have been made to the section on Classes of Parallelism to reflect the growth in reliance on GPUs and heterogeneous cores. There is now more coverage of distributed systems in the learning outcomes throughout.

Table 2: Architecture Topics

Topics	Core BLOOM level	Learning Outcomes and Teaching Suggestions (core)	Advanced Bloom Level	Learning Outcome (advanced)	Where Covered
Classes of Parallelism					
Taxonomy	C	Flynn's taxonomy, data vs. control parallelism, shared/distributed memory	C	Arch2: Flynn's taxonomy, data vs. control parallelism, shared/distributed memory (additional depth)	Systems; Arch2
Data parallelism					
Superscalar (ILP)	K	Describe opportunities for multiple instruction issue and execution (different instructions on different data)	A	Explain how superscalar works, how to schedule instructions for wider issue	Systems; Arch2
SIMD/Vector (e.g., AVX, GPU, TPU)	K	Describe uses of SIMD/Vector (same operation on multiple data items), e.g., accelerating graphics for games	C	Know the relationship of vector extensions to separate accelerators, and how they operate	Systems; Arch2
SIMD/Vector energy effects	K	Saves energy by sharing one instruction over many instruction executions, whether in parallel (SIMD) or pipelined (vector)	K	Arch2: Saves energy by sharing one instruction over many instruction executions, whether in parallel (SIMD) or pipelined (vector) (additional depth)	Systems; Arch2
Dataflow	N		K	Be aware of this alternative execution paradigm	Arch2
Pipelines					
Basic structure and organization	C	Describe basic pipelining process (multiple instructions can execute at the same time), describe stages of instruction execution	C	Arch2: Describe basic and more advanced pipelining process (including superpipelining and superscalar issue and commit, e.g., scoreboard, reservation stations, reorder buffer), describe stages of instruction execution (additional depth, such as forwarding, multithread handling, shelving)	Systems; Arch2

		Data and control hazards	K	Show examples of how one pipe stage can depend on a result from another, or delayed branch resolution can start the wrong instructions in a pipe that can reduce performance	C	Compilers: Instruction scheduling for modern pipe systems. Arch2: Pipeline effects including stalling, shelving, and restarting. Mechanisms for avoiding performance losses such as branch prediction, predication, forwarding, multithreading. DistSystems: understanding equivalent ideas as expressions of time and local versus global knowledge	Systems; Compilers; Arch2; DistSystems
		OoO execution, speculation	N		C	Be able to work through examples of Scoreboard, Reservation stations, Reorder buffers, difference between complete and commit, speculation (including security issues)	Arch2
		Streams (e.g., GPU)	K	Know that stream-based architecture exists in GPUs	C	Be able to describe basic GPU architecture with streaming multiprocessors, threads, warps, memory hierarchy	Systems; Arch2
	Control parallelism						
		MIMD	K	Identify MIMD instances in practice (multicore, cluster, e.g.), and know the difference between execution of tasks and threads			Systems
		Multi-Threading	K	Distinguish multithreading from multicore (based on which resources are shared)	C	Be able to explain the difference between coarse and fine grain multithreading, and simultaneous multithreading.	Systems; Arch2
		Scaling of Multithreading (e.g., GPU, IBM Power series)	N		K	Have an awareness of the potential and limitations of thread level parallelism in different kinds of applications	Arch2
		Multicore	C	Describe how cores share resources (cache, memory) and resolve conflicts	C	Be able to describe relationship of cores to memory hierarchy, reason for switching from increasing clock rate to increasing core count	Systems; Arch2
		Heterogeneous (e.g., GPU, security, AI, low-power cores)	K	Recognize that multicore may not all be the same kind of core (mix of organizations, instruction sets, high level explanation of benefits and costs))	C	Be able to describe performance benefits from integrating heterogeneous cores on a chip in context of applications with heterogeneous parallelism. Be able to cite reasons for some high performance accelerators to be on separate chips, due to need for greater silicon real estate	Systems; Arch2

		Heterogeneous architecture energy (small vs. big cores, CPU vs. GPU, accelerators)	K	Know that heterogeneity saves energy by using power-efficient cores when there is sufficient parallelism	C	Enumerate energy benefits from heterogeneity in architecture. Illustrate power/performance tradeoff. Discuss some combination of high variability with small feature sizes and reliability issues with lower voltages, redundancy techniques to enhance reliability, performance/reliability and power/reliability tradeoffs.	Systems; Arch2
	Shared memory						
		Symmetric Multi-Processor (SMP)	K	Explain concept of uniform access shared memory architecture			Systems
		Buses for shared-memory implementation	C	Be able to explain issues related to shared memory being a single resource, limited bandwidth and latency, snooping, and scalability	C	Arch2: Describe different arbitration schemes and tradeoffs. DistSystems: Explain need for symmetry breaking.	Systems; Arch2; DistSystems
		Broadcast (snooping)-based, Cache-Coherent Non-Uniform Memory Access (CC-NUMA)	N		K	Be aware that caches in the context of shared memory depend on coherence protocols, and understand idea of snooping (protocols are addressed in a separate topic)	Arch2
		Directory-based CC-NUMA	N		C	Arch2: Be aware that bus-based sharing doesn't scale, and directories offer an alternative, based on distributed memory hardware. DistSystems: see how shared memory can be extended to a loosely coupled systems with appropriate consistency model.	Arch2; DistSystems
	Distributed memory						
		Message passing (no shared memory)	N		K	Arch2: Shared memory architecture breaks down when scaled due to physical limitations (latency, bandwidth) and results in message passing architectures. ParProg: Be aware of the effect of network hardware support in context of e.g., MPI, PGAS implementation.	Arch2; ParProg

		Topologies	N		C	Algo2: Various graph topologies - linear, ring, mesh/torus, tree, hypercube, clique, crossbar and their properties (e.g., diameter, bisection width, etc.). DistSystems understand network as means of communication among distributed nodes, and effects of topology.	Algo2; ParProg; DistSystems
		Latency	K	Know the concept, implications for scaling, impact on work/communication ratio to achieve speedup. Awareness of aspects of latency in parallel systems and networks that contribute to the idea of time accumulating over dependent parts or distance	C	Algo2: Give examples of how communication latency contributes to complexity analysis of an algorithm. DistSystems: Show awareness of hardware aspects of latency in parallel systems and networks that contribute to the concepts of distributed time, and of local versus global knowledge. Differences between synchronous and asynchronous systems.	Systems; Algo2; DistSystems
		Bandwidth	K	Know the concept, how it limits sharing, and considerations of data movement cost			Systems
		Circuit and packet switching	N		C	Arch2: know that interprocessor communication can be managed using switches in networks of wires to establish different point-to-point connections, that the topology of the network affects efficiency, and that some connections may block others. Know that interprocessor communications can be broken into packets that are redirected at switch nodes in a network, based on header info. Networking: Be able to explain basic functionality and limitations within routers	Arch2; Networking
		Routing	N		C	Arch2: know that messages in a network must follow an algorithm that ensures progress toward their destinations, and be familiar with common techniques such as store-and-forward, or wormhole routing. Networking: Be able to explain wide area routing, packet latency, fault tolerance. DistSystems: see impact on distributed systems, see adaptive routing as instance of load balancing	Arch2; Networking ; DistSystems
Underlying Mechanisms							

		Caching	C	Know the cache hierarchies, and that shared caches (vs. private caches) result in coherency and performance issues for software	C	Arch2: Explain why coherence is complicated by synonym resolution within a virtual memory system. Know that TLBs are affected by sharing in a multicore context. Be aware of emerging non-volatile memory technology in the hierarchy. DistSystems: Show how network latency can be hidden by caching, but that it also introduces coherence issues	Systems; Arch2; DistSystems
		Atomicity	K	CS2: Show awareness of the significance of thread-safe data structures in libraries. Systems: Explain the need for atomic operations and their use in synchronization, both in local and distributed contexts.	C	Arch2: Understand implementation mechanisms for indivisible memory access operations in a multicore system OS: be able to use atomic operations in management of critical sections locally, in a multiprocessor. DistSystems: Be able to recognize or give examples of atomicity in a distributed system	CS2; Systems; Arch2; OS; DistSystems
		Consistency	N		C	Arch2: be able to explain implementation mechanisms for consistent views of data in sharing; OS: Be able to explain consistency from a process management perspective; ParProg: Be able to explain consistency from the perspective of language or library models; DistSystems: Be able to explain consistency from perspective of coping with high latency	Arch2; OS; ParProg; DistSystems
		Coherence	N		C	Arch2: examine protocols and their implementation and performance differences. ParProg, OS: describe how cores share cache and resolve conflicts. DistSystems: protocols for wide area coherence	Arch2; OS; ParProg; DistSystems
		● False sharing	N		C	Arch2: explain in context of excess coherence traffic. ParProg: learn how to avoid by structuring data and orchestrating access. DistSystems: explain how it can occur with remote data.	Arch2; ParProg; DistSystems

		Interrupts and event handling	K	CS2: know that event handling is as an aspect of graphical user interfaces. Systems: know that I/O is mostly interrupt-driven.	C	Arch2: know how interrupts are implemented, including the vector table, masking, and timers used in multiprocessing. Be able to explain handshaking protocols. OS: be able to work with interrupt handling, including from other processors and network sources, and the use of timer interrupts in multiprocessors. DistSystems: be able to use network interrupts in a distributed system and understand handshaking protocols for communications	CS2, Systems, Arch2; OS; DistSystems
		Handshaking	K	Know the difference between synchronous and asynchronous communication protocols, including context of multiple processors.	C	Arch2: able to explain handshaking protocols for buses. Networking: able to explain handshaking in network protocols. DistSystems: be able to use handshaking protocols for communication	Systems; Arch2; Networking; DistSystems
		Process ID, System/Node ID	K	Explain need for a process identifier to enable interprocess communication, and for a node identifier in initialization of multiprocessor systems	C	Arch2: know how process ID is implemented, see need for node ID for symmetry breaking. ParProg: be able to use process ID for communication, node ID for symmetry breaking in programs. DistSystems: know how node ID is a key element of local knowledge	Systems; Arch2; ParProg; DistSystems
		Virtualization Support	N		C	Arch2, OS, DistSystems: Enumerate pros and cons of virtualization for security, load balancing, quality of service management, and/or issues of supporting with good performance and security	Arch2; OS; DistSystems
Floating Point Representation				These topics are supposed to be in the ACM/IEEE core curriculum already – they are included here to emphasize their importance, especially in the context of PDC for large problems.			
		Range	K	Explain why range is limited, implications of infinities			CS1; CS2; Systems
		Precision	K	How single and double precision floating point numbers impact software performance, basic ways to avoid loss of precision in calculations			CS1; CS2; Systems

		Rounding issues	N		C	Arch2: Explain differences in rounding modes.
						Algo2: Give examples of accumulation of error and loss of precision.
						ParProg: Explain need for numerical analysis to ensure that accumulation of error in long chains of calculation over large data sets is controlled
							Arch2;
							Algo2;
							ParProg

		Error propagation	K	Able to explain NaN, Infinity values and how they affect computations and exception handling			CS2
		IEEE 754 standard	K	Representation, range, precision, rounding, NaN, infinities, subnormals, comparison, effects of casting to other types			CS1, CS2, Systems
Performance Metrics							
		Instructions per cycle	C	Explain pros and cons of IPC as a performance metric, various pipelined implementations	C	Describe impact of multithreading on IPC calculation	Systems; Arch2
		Benchmarks	K	Awareness of various benchmarks (such as LinPack, NAS parallel) and how they test different aspects of performance in parallel systems			Systems
		● Bandwidth benchmarks	N		K	Be aware that there are benchmarks focusing on data movement instead of computation, and that there are different architecture aspects that contribute to bandwidth, and that bandwidth is with respect to the source and destination points across which it is measured	Arch2
		Memory bandwidth	N		C	Be able to explain significance of memory bandwidth with respect to multicore access, and different contending workloads, and challenge of measuring	Arch2
		Network bandwidth	N		C	Know how network bandwidth is specified and explain the limitations of the metric for predicting performance, given different workloads that communicate with different contending patterns	Arch2
		Peak performance	C	Explain what peak performance is and how it is rarely valid for estimating real performance, illustrate fallacies			Systems

		● MIPS/FLOPS	K	Know meaning of terms			Systems
		Sustained performance	C	Know difference between peak and sustained performance, how to define, measure, different benchmarks			Systems
Power							
		Power, Energy	C	Be able to explain difference between power (rate of energy usage) and energy			Systems
		Large scale systems, distributed embedded systems	N		K	Know about active power states and power reduction of CPUs using dynamic voltage and frequency scaling (DVFS), idle power management using sleep states. In an advanced architecture course, could cover link power reduction of multilane serial links and power management as it relates to large scale systems and low-power IoT type devices	Arch2
		Power density	N		K	Be aware that power density can create challenges in thermodynamics of chip designs, for cooling technologies, or for the supply of power to a compute unit (e.g. a rack). Power density challenges for modern architectures	Arch2
		Static vs. dynamic power	N		C	Grasp the concept of how some energy usage is relatively constant, and some varies with compute load or data content (1's and 0's). Explain the significance of leakage current as chip feature size shrinks and how scaling up core count may require reduced clock rate to compensate	Arch2
		Clock scaling / clock gating / Power gating /thermal controls	N		K	Know that clock scaling/gating is a technology for reducing the power used for latches and other storage units that do not need to be changed. Know that modern systems have thermal sensors that guide clock scaling	Arch2
		Power efficient HW design: simpler cores, short pipelines	N		C	Explain the power-efficiency advantages of simpler designs. To apply, could engage in an exercise using out-of-order core vs. in-order core, providing hypothetical performance and energy values.	Arch2

Scaling (HPC, Big Data)							
		Reliability and fault tolerance issues (cross cutting topics of current curriculum)	N		K	Large-scale parallel/distributed hardware/software systems are prone to components failing but the system as a whole needs to work.	Arch2
		Hardware support for data bound computation	N		C	May include caching, prefetching, RAID storage, high performance networks such as Infiniband. Know that modern architectures are starting to provide hardware support for operations that are crucial for data science. May include GPU collective operations, memory access coalescing	Arch2
		Hardware limitations for data bound computation	K	Comprehend the hardware limitations in the context of Big data and their most relevant v's (volume, velocity)	C	Arch2: considerations of memory system bandwidth, latency hiding, network bandwidth, direct interprocessor links (e.g., QPI, GPUDirect). ParProg: be able to use tools to identify data transfer bottlenecks.	Systems; Arch2; ParProg
		Pressure imposed by data volume	K	Know of the limitations in terms of storage, memories, and filesystems when dealing with very large data volumes	C	Arch2: Compare and contrast limitations of different storage technologies. ParProg: be aware of architectural aspects of large systems that affect structuring computation to enable load balancing, and optimizations for throughput.	Systems; Arch2; ParProg
		Pressure imposed by data velocity	K	Know of the limitations in bandwidth, memory, and processing when processing very fast data streams. Include networking and reliability issues	C	Arch2: Explain the need to balance system resources to avoid bottlenecks in processing fast data streams. ParProg: Explain architectural aspects of data velocity issues of load balancing, jitter, and skew in coordinating flows of data.	Systems; Arch2; ParProg
		Cache growth with scaling out	N		K	Know the difference between scaling up and scaling out. As the number of nodes scales, total cache space increases, which can allow partionable problems to also scale more effectively (giving appearance of super-linear speedup), and may impact inter-node communication	Arch2

		Cost of data movement across memory hierarchies	C	Comprehend the difference in speed vs cost (latency, energy) of accessing the different memory hierarchies, and of changing their sizes, in the context of multiple processors and hierarchies	C	Arch2: Enumerate factors affecting bandwidth, latency, power. ParProg: Describe impact on synchronization, load balancing, tools for measurement. How increased core counts may be seen as a means to increase total cache space in a large system.	Systems; Arch2; ParProg

6. Programming Topics

6.1 Rationale

The material is organized into three subtopics: Paradigms and notations, correctness, and performance/energy. We discuss these in separate sections below. A prerequisite for coverage of much of this material is some background in conventional programming. Even though we advocate earlier introduction of parallelism in a student's programming experience, basic algorithmic problem-solving skills must still be developed, and we recognize that it may be easier to begin with sequential models. Coverage of parallel algorithms prior to this material would allow the focus to be exclusively on the practical aspects of parallel and distributed programming, but they can also be covered at the same time as necessary and appropriate. Parallel software development can be taught using many different languages and tools, including Java, C, C++, Python, OpenMP, CUDA, MPI, and many others.

Paradigms and Notations: There are different approaches to parallel programming. These can be classified in many different ways. Here we have used two different ways of classifying the models. First, we classify the paradigms by the target machine model: *SIMD* (single instruction multiple data) is the paradigm in which the parallelism is confined to operations on (corresponding) elements of arrays. This linguistic paradigm is at the basis of the Intel Advanced Vector Extensions (AVX) or IBM Vector Scalar Extension (VSX) macros, some database operations, some operations in data structure libraries, and the languages constructs used for vector machines. *Shared-memory* is the paradigm of OpenMP and Intel's Thread Building Blocks, among other examples. *Distributed memory* is the paradigm underlying message passing and the MPI standard. A hybrid model is when any of the previous three paradigms co-exist in a single program. The logical target machine does not have to be identical to the physical machine. For example, a program written according to the distributed memory paradigm can be executed on a shared-memory machine and programs written in the shared-memory paradigm can be executed on a distributed memory machine with appropriate software support (e.g., Intel's Cluster OpenMP). More loosely coupled models are also addressed, including client/server, peer-to-peer, and big data models.

A second way to classify programming approaches is according to the mechanisms that control parallelism. These are (mostly) orthogonal to the first classification. For example, programs in the SPMD (single program multiple data) paradigm can follow a distributed-memory, shared-memory or even the SIMD model from the first classification. The same is true of programs following the data parallel model. The task spawning model can work within a distributed or shared-memory paradigm. The parallel loop form seems to be mainly used with the shared-memory paradigm, but some models/languages have merged the loop model with the distributed memory paradigm (e.g., MapReduce). The recent trend toward accelerators (e.g., GPUs) is also included here. 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, AVX macros, OpenMP, and MPI.

Correctness and Semantics: This set of topics outlines the material needed to understand the behavior of parallel programs beyond the fact that there are activities that take place (or could take place) simultaneously. Material in this section covers three main topics as well as the methods and tools necessary to detect and troubleshoot defects. The main topics are Tasking, Synchronization and Memory models. In this context, tasking refers to the means to create threads on multiple cores and assign work to them, either through implicit or explicit assignment (e.g., OpenMP vs. POSIX threads). Synchronization introduces the concept of critical sections of code that depend on order to execute properly. Material in this section includes critical regions in code as well as producer-consumer models. The third section covers different Memory models explaining the relationship and tradeoffs between strict and relaxed memory models. As many programming languages have their own model, only the basic ideas are expected to be covered. Finally, the section deals with Concurrency problems and the tools to detect them. Common defects such as deadlock, starvation and race conditions are explained while tools like Eraser or various tools from Intel's Parallel Toolkit can be covered as a means of detecting the errors.

Performance and Energy: The final group of topics is about performance and energy issues - how to organize the computation and the data for the different classes of machines, and how to deal with minimizing energy usage. Topics in this section are divided in four categories: Computation, Data, Tools and Metrics, and Power/Energy Efficiency. The first two categories (i.e., Computation and Data) refer to parallel programming aspects that influence performance, either in the form of task decomposition or by accessing remote data. The third topic introduces the notion of performance metrics and the ways in which they can be collected. In more detail, *Computation* includes task decomposition strategies and how tasks can be assigned to different threads/processes. This section includes performance effects of load balancing and its contributing factors. such as scheduling, failures, and distribution delays. *Data* includes a variety of topics, from data representation and its effect on performance and energy, to data locality, and performance tradeoffs of laying out data and accessing data remotely. This section introduces different storage and organization paradigms, as well as issues arising from distribution and replication, such as consistency and atomicity of operations. *Tools and Metrics* covers ways in which performance is measured and evaluated, the laws measuring performance in a parallel setting, and finally performance in relation to *Energy/Power* consumption.

6.2 Updates from version 1.0

We highlight the changes to the curriculum guidelines relative to version 1.0 of this document. The main changes are related to adding topics from new aspects, including big data, distributed computing, and energy. Many of those topics are suitable for upper level classes in parallel programming, distributed systems, databases and others, but some topics have influenced the learning outcomes for core classes. The energy topics can be addressed in the core Systems class. One additional change is the addition of programming for accelerators, such as GPUs, which can be introduced in a CS2 or Systems class and addressed more deeply in an upper level Parallel Programming class.

Table 3: Programming Topics

Topics	Core Bloom Level	Learning Outcomes and Teaching Suggestions (core)	Advanced Bloom Level	Learning Outcome (advanced)	Where Covered
Parallel Programming Paradigms and Notations					
By the target machine model					
Concurrency and Parallelism	C	Understand concurrency is an algorithmic property; it exposes potential for parallelization. If concurrency is present in an algorithm, it can be parallelized, without concurrency there is no scope for parallelization. Concurrency can be present in a sequential program, parallelization takes advantage of concurrency to increase performance.			CS1; CS2; DS/A
SIMD	K	Understand common vector operations including element-by-element operations and reductions.			CS2: Systems
• Processor vector extensions	K	Know examples - e.g., Intel AVX or Power VSX macros	C	Understand examples from Intel/Power vector instructions	Systems; Arch2
• Array language extensions	N		A	Know how to write parallel array code in some language (e.g., Fortran95, Intel's C/C++ Array Extension[CEAN])	ParProg
Shared memory	A	Be able to write correct thread-based programs (protecting shared data) and understand how to obtain speed up.			CS2; DS/A
• Language parts or extensions	K	Know about language extensions for parallel programming. Illustrate with examples from Cilk (spawn/join), Java (Java threads), or other languages.			

		● Compiler directives/ pragmas	C	Understand what simple directives, such as those of OpenMP, mean (parallel for, concurrent section), show examples			
		● Libraries	C	Know one in detail, and know of the existence of some other example libraries such as Pthreads, Pfunc, Intel's TBB (Thread building blocks), Microsoft's TPL (Task Parallel Library), C++ threads, etc.			
		Distributed memory	K	Know basic notions of messaging among processes, different ways of message passing, collective operations			Systems; DS/A
		● Message passing	N		C	Know about the overall organization of a message passing program as well as point-to-point and collective communication primitives (e.g., MPI)	ParProg
		● PGAS languages	N		C	Know about partitioned address spaces, other parallel constructs (e.g., UPC, CoArray Fortran, Chapel)	ParProg
		Client Server and Peer-to-Peer models	C	Know notions of invoking and providing services (e.g., RPC, RMI, web services) - understand these as concurrent processes; know about network model of distributed computing (e.g., sockets); know that in distributed systems such handshaking interaction is crucial for the efficient communication between asynchronous processes	A	Be able to program a basic client/server and/or P2P interface	Systems; Networking
		Big Data Technology Stack	N		A	Understand the Big data technology stack and its layered architecture. Be able to write code (e.g., in Python, R) using some of the tools that facilitate data storage, organization, management, and/or analysis within a Big Data stack	ParProg; DistSystems

		Hybrid	K	Know the notion of programming over multiple classes of machines simultaneously (CPU, GPU, TPU, etc.)	A	Be able to write correct programs using two programming paradigms: e.g., shared memory and GPU (OpenMP+CUDA), Distributed and shared memory (MPI+OpenMP), Distributed memory and GPU (MPI+CUDA)	Systems; ParProg
	By the control statement						
		Task/thread spawning	A	Be able to write correct programs with threads, synchronize (fork-join, producer/consumer, master/worker, etc.), use dynamic threads (in number and possibly recursively) thread creation - (e.g., Pthreads, Java threads, etc.) - builds on shared memory topic above			CS2; DS/A
		Event-Driven Execution	K	Know about the need for event-driven execution; possible approaches to implementing it. Know about the notion of causality among events (e.g., remote file access, GUI). These effects may be easier to discuss in the context of distributed systems.	A		CS2; DistSystems
		SPMD	C	Understand how SPMD program is written and how it executes		Be able to write an SPMD program and understand how it executes	
		• SPMD notations	C	Know the existence of highly threaded data parallel notations (e.g., CUDA, OpenCL), message passing (e.g., MPI), and some others (e.g., Global Arrays, BSP library)	A		CS2; DS/A; ParProg
		Data parallel	A	Be able to write a correct data-parallel program for shared-memory machines and get speedup, should do an exercise.		Understand relation between different notations for data parallel: Array notations, SPMD, and parallel loops. Builds on shared memory topic above.	
		• Parallel loops for shared memory	A	Know, through an example, one way to implement parallel loops, understand collision/dependencies across iterations (e.g., OpenMP, Intel's TBB)	A		CS2; DS/A; Lang
		• Data parallel for distributed memory	N		K	Know data parallel notations for distributed memory (e.g., UPC, Chapel, Co-Array Fortran)	ParProg

		• MapReduce	K	Understand how problems can be solved by mapreduce, and how algorithms can be written using map and reduce	A	Solve problems using mapreduce, and write algorithms using map and reduce	CS2; Lang; ParProg
		Offloading to accelerators	K	Know about running parts of applications on accelerators (e.g., GPU, TPU, FPGA)			CS2; Systems
		• Accelerator notations	N		A	Be able to write an accelerator program that takes advantage of the hardware (e.g., CUDA, OpenACC, OpenMP 4.5 or above, OpenCL, TensorFlow)	ParProg
		Functional/logic languages	N		K	Understanding advantages and disadvantages of very different programming styles (e.g., Parallel Haskell, Parlog, Erlang)	ParProg
Semantics and correctness issues							
	Tasks and threads		A	Be able to write parallel programs that create and assign work to threads/processes,, in at least one parallel environment (e.g., OpenMP, Intel TBB, pthreads, etc.)	A		CS2; DS/A; Systems; Lang
	Synchronization		A	Be able to write shared memory programs with critical regions, producer- consumer communication, and get speedup; know the notions of mechanisms for concurrency (monitors, semaphores, etc.)			CS2; DS/A; Systems
		Critical regions	A	Be able to write shared memory programs that use critical regions for synchronization			
		Producer-consumer	A	Be able to write shared memory programs that use the producer-consumer pattern to share data and synchronize threads			
		Handshaking	N		A	Analyze handshaking protocols and derive performance bounds (e.g., response time in sliding window, TCP connection management)	Networking; DistSystems

	Concurrency issues		C	Understand the notions of deadlock (detection, prevention), race conditions (definition), determinacy/non-determinacy in parallel programs (e.g., if there is a race condition, the correctness of the output may depend on the order of execution)			DS/A; Systems
		Deadlock/Livelock	C	Understand what deadlock and livelock are, and methods for detecting and preventing them; also cast in terms of distributed systems			
		Starvation	C	Understand how starvation (of a thread or process) can occur, in context of an example (e.g., dining philosophers)			
		Race condition	C	Know what a race condition is, and how to use synchronization to prevent it			
		Distributed Data Structures and Applications	N		C	Understand synchronization in the context of data structures; correctness in a concurrent context	ParProg; DistSystems
		Tools to detect concurrency defects	N		K	Know the existence of tools to detect race conditions (e.g., Eraser) and debugging (e.g., Intel Parallel Toolkit)	ParProg
	Memory models		N		C	Know what a memory model is, and the implications of the difference between strict and relaxed models (performance vs. ease of use)	ParProg
		Sequential consistency	N			Understand semantics of sequential consistency for shared memory programs	
		Relaxed consistency	N			Understand semantics of one relaxed consistency model (e.g., release consistency) for shared memory programs	
		Consistency in distributed transactions	N		C	Recognize consistency problems. Know that consistency is an issue in transactions issued concurrently by multiple agents. Implement transaction commit protocols in databases.	DB; DistSystems

Performance and Energy issues							
	Computation		C	Understand the basic notions of static and dynamic scheduling, mapping and impact of load balancing on performance			
		Computation decomposition strategies	C	Understand different ways to assign computations to threads or processes			CS2; DS/A
		• Owner computes rule	C	Understand how to assign loop iterations to threads based on which thread/process owns the data element(s) written in an iteration			
		• Decomposition into atomic tasks	C	Understand how to decompose computations into tasks with communication only at the beginning and end of each task, and assign them to threads/processes			
		• Decomposition into Map and Reduce tasks	K	Know that divide and conquer can be expressed and programmed as a Map and Reduce decomposition	A	Understand how to decompose computation using the Map Reduce paradigm. Be able to reason about computation speed ups from this decomposition and the communication tradeoffs resulting from reduction(s). Be able to produce code expressing this decomposition	CS2; ParProg
		• Work stealing	N		C	Understand one way to do dynamic assignment of computations	ParProg
		• Offloading onto an accelerator	N		C	Understand when it is worthwhile to decompose onto an accelerator (e.g., GPU, TPU, FPGA)	ParProg
		Program transformations	N		C	Be able to perform simple loop transformations by hand, and understand how that impacts performance of the resulting code (e.g., loop fusion, fission, skewing, blocking)	Compilers; ParProg

		Load balancing	C	Understand the effects of load imbalances on performance, and ways to balance load across threads or processes			DS/A; Systems
		Scheduling and mapping	C	Understand the importance of a programmer, compiler and/or runtime system mapping and scheduling computations to threads/processes, both statically and dynamically	A	Can apply a static and a dynamic strategy for mapping processes of large scale parallel programs onto processors that optimizes performance through reduction of communication delays	DS/A; Systems; ParProg
		Effect of timing failures/delay in distributed systems	K	Understand that a failure in one node can cause a global failure in a distributed system. For example, one could use waiting on a non-terminating program to illustrate a failure scenario in distributed systems (e.g., in the context of consensus).			CS2
		• protocol timeout protections	N		A	Understanding the use of timeouts in situations with a high probability of error	Networking
	Data			Understand impact of data distribution, layout and locality on performance; notion that transfer of data has fixed cost plus bit rate (irrespective of transfer from memory or inter-processor); know false sharing and its impact on performance (e.g., in a cyclic mapping in a parallel loop) in ParProg;	C		DS/A; Lang; ParProg
		Data distribution	N		C	Understand what block, cyclic, and block-cyclic data distributions are, and what it means to distribute data across multiple threads/processes	ParProg
		Data layout	C	Know how to lay out data in memory to get improved performance and energy (memory hierarchy in shared memory parallel system)			DS/A; Systems
		• False sharing	K	Know that for cache coherent shared memory systems, data is kept coherent in blocks, not individual words, and how to avoid false sharing across threads of data for a block	C	Be aware of false sharing, able to give examples where it occurs, and understand why it happens	Systems; ParProg

		● Energy impact	K	Know the energy cost of loading data into memory from secondary storage (and writing out modified data to secondary storage)			Systems
		Data Representation					
		● Floating point and integer precision (64-bit, 32-bit, and 16-bit or less)	K	Power savings using smaller data representation 64-bit vs. 32-bit vs. 16-bit floating-point precision, 32-bit vs. 16-bit integer). For example, machine learning on GPUs is driving lower (16-bit) floating-point precision.			Systems
		Data locality	K	Know what spatial and temporal locality are, and how to organize data to take advantage of them			DS/A; Systems
		● Performance impact of data movement	K	Know the performance cost of moving data to secondary storage for big data analysis, distributed vs centralized analysis. Be aware of specific mechanisms that take advantage of locality (e.g., in-situ processing)	C	Understand performance costs of data locality with respect to various metrics, and be able to contrast mechanisms like in-situ vs. in-transit vs. offline processing	Systems; ParProg; DistSystems
		● Structured vs unstructured data	K	Know the differences and tradeoffs between these data representations	A	Be able to build solutions for the different types of data	DS/A; Databases
		● Graph representations and databases	K	Know of graph representations of data to facilitate graph analysis.	C	Understand performance gains due to graph-specific mechanisms versus other more general data representations	DS/A; Databases
		Data handling and manipulation	N		C	Understand the differences and performance impact of data formatting and storing mechanisms. Be able to implement adequate solutions for the different types of storing	DistSystems; Databases
		● Distributed databases	N		C	Comprehend the principles behind distributed databases and the motivation of tradeoffs to support scalability	DistSystems; Databases
		● NoSQL databases	N		A	Comprehend how NoSQL databases enable scalable data manipulation, include exemplars to become familiar with some of them (e.g., MongoDB, Hive)	Databases

		• Eventual consistency vs ACID	N		C	Comprehend how replication enables scalability in databases but transactions lose their ACID properties	Databases
		Distributed file systems	K	Be aware of existence of distributed file systems and common examples of where they are used and why	K	Comprehend the basic principles of how distributed file systems work, their scalability benefits, and performance and reliability problems	Systems; Operating Systems; DistSystems; ParProg
		• Replicated file systems	N		C	Know of distributed file systems such as HDFS and its replication and fault tolerance mechanisms. ParProg be able to use a DFS.	DistSystems; ParProg
		• Key-value storage systems	N		K	Understand the concept of key-value storage. ParProg be able to use an appropriate API.	DistSystems; ParProg
	Tools and metrics						
		Performance monitoring tools	K	Know of tools for runtime monitoring (e.g., gprof, perf, Intel Performance Toolkit, TAU)			DS/A; Systems
		Performance metrics	C	Know the basic definitions of performance metrics (speedup, efficiency, work, cost), Amdahl's law; know the notion of scalability			CS2; DS/A
		• Speedup	C	Understand how to compute speedup, and what it means			
		• Efficiency	C	Understand how to compute efficiency, and why it matters			
		• Parallel Scalability	C	Understand that speedup and efficiency is a single point of measure for a particular problem size and number of processes/threads. These metrics change as problem size and/or number of processes/threads vary. Understand that scalability is a metric that measures how speedup varies as problem size and/or number of processes/threads vary.			

		• Amdahl's law	C	Understand that speedup is limited by the sequential portion of a parallel program, if problem size is kept fixed			
		• Gustafson's Law	K	Know the idea of weak scaling, where problem size increases as the number of processes/threads increases			
	Power/Energy efficiency						
		Power-latency tradeoff	K	Familiar with the notion that problem decompositions (including their granularity), and active/idle states (e.g., including modulation of CPU frequency) may be exploited to adjust balance among throughput, latency, and energy consumption.			Systems
		Energy efficiency vs. load balancing	K	Aware that unbalanced work decomposition and communication congestion can prolong computation and reduce energy efficiency.			Systems
		Active power management methods	N		A	Aware that systems expose various execution parameters (e.g., P states on Intel) and have examined the effect of modulating at least one to optimize performance or energy consumption.	ParProg; Arch2
		Idle power management methods	N		A	Aware that architectures and OSs provide various interfaces that enable computational units to be passivated (e.g., C states) and have examined tradeoffs (e.g., reduced subsystem power consumption v. increased latency) involved in exploiting at least one of them.	ParProg; Arch2
		Power consumption of parallel programs	N		K	Aware that optimal energy efficiency may not be achieved through aggressive reduction of CPU clock frequencies and exploitation of sleep modes due to increased execution time and static components of system power consumption.	ParProg

Security							
		Security protocols	N		A	understand IPsec basics	Networking

7. Algorithms Topics

7.1 Rationale

Similar to other areas, the Algorithms topics fall into topics covered at various levels in core courses and topics that are important but may be covered in non-core courses. The topics designated for core courses may be introduced alongside their traditional sequential counterparts without adding a significant amount of instruction time.

Many of the topics build on sequential algorithm ideas that are already covered in a typical curriculum; the list of topics, in many cases, highlights the opportunity to use this sequential coverage to introduce parallel and distributed algorithmic concepts. These topics are organized under three broad sections. The **Parallel/Distributed Models and Complexity** section contains foundational topics aimed at equipping the students with the basic computational considerations and skills needed for designing and analyzing parallel algorithms. The **Algorithmic Techniques** section covers recurrent themes or constructs that are generally useful as building blocks in designing a wide variety of parallel algorithms. In the **Algorithmic Problem** section, we include some basic problems for which we feel that learning parallel algorithms would be valuable for almost all CS/CE students.

Parallel/Distributed Models and Complexity: The goal of this section is to introduce the foundational aspects of parallel and distributed algorithms, including computational and communication models, and performance metrics; we build on this foundation to develop other topics. We walk through these considerations, placing them in the context of parallel and distributed algorithms.

One of the most fundamental differences between sequential and parallel/distributed algorithms is along the time dimension. As a performance metric, time, which can be shared by concurrent processes, is very different in PDC compared to sequential algorithms. In contrast, energy, area/hardware and memory (to within reuse), behave similarly in sequential and parallel or distributed contexts. A basic manifestation of time in parallel and distributed algorithms is in the form of asynchrony, with associated issues, such as races and hazards, data consistency, and event-driven execution. The efficient use of resources to harness the time-benefits of parallelism brings up measures such as speedup and scalability, and approaches including scheduling and load-balancing.

Another important difference between the sequential and parallel/distributed context is the communication structure between processing elements. Algorithmic strategies must account for the advantages and limitations of a given communication fabric. For this, we suggest the introduction of a relatively simple, parallel and/or distributed computational model, a “model of choice,” on which key algorithmic concepts can be developed without getting into implementation details. A PRAM-like model is useful to bring out the raw parallelism in an approach, whereas a completely-connected network model may address aspects of communication without topological constraints. Another possible model is one of distributed Big Data, where the impracticality of moving data plays a prominent part in algorithm design.

The topics identified in this section emphasize foundational aspects of parallel and distributed algorithms and act as stepping stones to topics in Sections 6.2 and 6.3. They have also been selected to not overly depend on technological particulars, but rather emphasize core ideas with greater longevity. Our choices reflect a combination of "persistent truths" and "conceptual flexibility." Our suggestions strive to impart both an understanding of how one reasons rigorously about the expenditure of computational resources (time, memory, energy), and an appreciation of fundamental computational limitations that transcend the details of particular platforms.

Algorithmic Techniques: This section acknowledges the folk wisdom that contrasts giving a fish and teaching how to fish. Algorithmic techniques lie in the latter camp. We have attempted to enumerate a variety of techniques, patterns, or kernels whose algorithmic utility has been demonstrated over the course of decades. Many of these can be viewed as algorithmic analogues of high-level library functions or methods. The parallel kernels or patterns in our list can be viewed as control structures that would be frequently encountered while developing parallel algorithms for a diverse set of computational tasks or problems, and are readily available on a broad range of parallel and distributed computing platforms.

Included are some commonly-used global computations such as broadcast and reduction, which may appear trivial in a serial environment, but must be recognized as special communication kernels in a parallel environment. Due to their global nature, their optimal implementation could be a critical determinant of the complexity/performance of the parallel algorithm employing them. Other examples include parallel prefix (scan), gather, and scatter.

This section also includes task and data decomposition paradigms such as divide-and-conquer, recursion, blocking and striping, and load balancing. Modern approaches such as MapReduce are introduced as a problem-solving technique in distributed systems. Additionally, this section touches on pervasive topics of synchronization, mutual exclusion, and conflict resolution.

We believe that students with the appropriate level of knowledge of the building blocks covered in this section would be reasonably equipped to design and analyze basic parallel algorithms, some of which are included in the next section.

Algorithmic Problems: This section includes problems for which we recommend designing, analyzing, and in some cases, implementing parallel algorithms. It highlights important problems that play such key roles in a variety of computational situations that we view them as essential to an education about parallel and distributed computing. These problems can serve as examples with which to illustrate the algorithmic techniques from the previous section. Some of the problems are implementations of techniques mentioned in the previous section.. For example, parallel-prefix (scan) appears in both the Algorithmic Techniques and Algorithmic Problems sections. The former recommendation is to teach scan as a primitive for use in solving other problems and the latter is to teach how the scan itself can be implemented. In many cases, these may be taught together, but having separate recommendations highlights the two aspects and allows for courses or curricula to treat them separately.

Also included in this section is a list of algorithmic problems from which we recommend choosing one or two for a relatively in-depth design, analysis, and/or implementation. The problems can be chosen based on students' and instructor's interest or fit with the existing algorithms curriculum. We believe that every modern CS/CE undergraduate student should encounter an in-depth experience with the design and analysis of at least one parallel or distributed algorithm. The problem or the algorithm chosen for deeper investigation may also be used to explore one or more of the earlier topics. For example, matrix multiplication could be used to study the impact of different decomposition schemes on speedup and efficiency, or alternatively, search could be used to explore dynamic load balancing.

7.2 Updates from version 1.0

The number of core Algorithms topics recommended has been reduced and the recommended level of coverage has been lowered for many of the retained topics, thus reducing the total additional effort required to infuse algorithmic PDC concepts into the core CS/CSE curriculum. However, there are two noteworthy additions. The first is the recommendation around a model of choice (MOC); the MOC admits development of key algorithmic concepts, while abstracting away from implementation details. Algorithmic analysis requires agreement on the allowed operations and their cost so it is important to define a concrete model. That said, we recognize that different models are appropriate for different kinds of algorithms and thus leave the choice of which to use up to the instructor. The second is the recommendation of a capstone deeper algorithmic experience, as described in Section 7.3 above.

Other changes to the recommendations in the Algorithms area relative to version 1.0 of this document include: (1) Updating topics to reflect current issues and trends (e.g., energy as a cost metric) (2) Streamlining the table and reducing the total number of topics, (3) Reorganizing and renaming the Algorithmic Techniques section, (4) Updating most learning outcomes, (5) Adding learning outcomes and courses.

Table 4: Algorithms Topics

Topics	Core Bloom level	Learning Outcomes and Teaching Suggestions (core)	Advanced Bloom Level	Learning Outcomes (advanced)	Where Covered
Parallel and Distributed Models and Complexity		Be exposed to the foundational aspects of parallel and distributed algorithms, including computational and communication models, and performance metrics			

	Concurrency, Asynchrony, Dependencies, and Nondeterminism		C	Qualitatively understand the notion of concurrency, asynchrony, dependencies, and nondeterminism through one or more every day examples illustrating simultaneous events with dependencies. The examples can be non-computational in CS1; for example, preheating the oven and preparing the batter for a cake can proceed concurrently and asynchronously to save time, but both these events must finish before baking starts (dependency). Computational examples of the appropriate level can be used in CS2. The goal for the instructor is to develop a baseline of ideas upon which to build the PDC concepts.			CS1; CS2
	Costs of computation			Be exposed to the broad concepts of parallel time and space complexity			
		Asymptotics	C	Be able to describe upper (big-O) and lower bounds (big-Omega,) in the context of PDC, understanding that the functions whose order is being analyzed may have an additional variable related to the number of processing elements in the PDC context. For example, serial run time for simple matrix multiplication is Theta(n^3), and the parallel run time may be Theta(n^3/p), where the number of processing elements p is the additional variable.			DS/A; CS2
		Time (number of operations) complexity	C	Recognize time as a fundamental computational resource that can be influenced by parallelism			DS/A
		Work	C	Understand the definition of computational work and how it is different from time. Be able to observe its impact on complexity measures such as time, speedup, energy consumption, etc.			DS/A
		Space/Memory	C	Recognize space/memory in the same manner as time.			DS/A

		Memory and Communication complexity	C	Understand that data movement (such as memory accesses or communication) may take more wall-time and energy than computations. Understand also that in certain distributed "big data" applications, moving data is cost prohibitive.			DS/A
	Performance Metrics			Be exposed to a variety of computational costs that are affected by PDC.			
		Speedup	C	Recognize the use of parallelism either to solve a given problem instance faster or to solve larger instance in the same time. Understand and be able to explain why there's an upper limit on speedup for a given problem of a given size (Amdah's law).			DS/A
		Efficiency, Scalability, Throughput	K	Know about the notions of efficiency, strong and weak scaling, and throughput.	C	Comprehend via several examples that having access to more processors does not guarantee faster execution (for example Amdahl's Law)	DS/A; Algo2
	Tradeoffs					Recognize the inter-influence of various cost measures	
		Time vs. space	N		C	Understand through an illustration in the context of, perhaps BigData, that not all information may be saved for future reference. This has an impact on the time needed to perform the computation. Observe several examples of this prime cost tradeoff; lazy vs. eager evaluation supplies many examples. Observe that recomputing a result may sometimes be more energy efficient than storing and retrieving the result.	Algo2; OS

		Power vs. time	N		C	Understand through at least one example this prime cost tradeoff (the literature on "VLSI computation" --- e.g., the footnoted books[1] [2] --- yield many examples). For example, trade power-intensive communication for extra computation, even if the latter takes more time. Understand that an imbalanced load may be better from the power consumption point of view as some processes could be hibernated. Understand that recomputing a result may be more power efficient than sending the computed result to multiple nodes (for example computing the join in a database context).	Algo2; ParAlgo; DistSystems
		Power vs. precision	N		C	Understand that power savings using smaller data representation 64-bit vs. 32-bit vs. 16-bit floating-point precision, 32-bit vs. 16-bit integer). Machine learning is driving lower (16-bit) floating-point precision.	Algo2; OS
		Isoefficiency (Work, Speedup, Efficiency tradeoffs)	N		C	Understand the idea of how to increase problem size as a function of the number of processes/threads to keep efficiency the same	ParProg; Algo2
	Model-based notions			Recognize that architectural features can influence amenability to parallel cost reduction and the amount of reduction achievable			
		Notions from complexity-theory		Understand (via examples) that some computational notions transcend the details of any specific model			
		● Model(s) to abstract from architectural details (for example PRAM (shared mem) and/or completely connected (network))	K	Understand concurrency basics without the trappings of real systems (routing, data alignment etc.). Recognize the PRAM as embodying the simplest forms of parallel computation: Embarrassingly parallel problems can be sped up easily just by employing many processors. Recognize how a completely connected network abstracts away from routing details. Recognize the difference between the model(s) and real systems. Such a MODEL of CHOICE (MoC) is assumed to be adopted by the	C	Observe examples of "model-independent" algorithms that ignore the details of the platform on which they are executed. Recognize architecture independence as an important avenue toward understanding the core ideas of parallelism. Explore fast algorithms (O(1) and O(log n) time, such as search and prefix computation); explore read/write rules (concurrent/exclusive). Explore communication	DS/A; Algo2

				instructor on which PDC concepts would be discussed.		centric algorithms (broadcast, multicast, gossip)	
		● BSP	N		K	Be exposed to higher-level algorithmic abstractions that encapsulate synchronization and other aspects of real architectures. BSP would be a good option to introduce a higher level programming model and higher-level notions. Remark that this abstraction has led to programming models.	Algo2
		● Simulation/ emulation	N		K	See simple examples of this abstract, formal analogue of the *virtual machines* that are discussed under programming topics. It is important to stress that (different aspects of the same) central notions of PDC can be observed in all four of our main topic areas.	Algo2
		Notions from scheduling		Understand how to decompose a problem into tasks			
		● Dependencies	A	Understand how dependencies constrain the execution order of sub-computations (thereby lifting one from the limited domain of "embarrassing parallelism" to more complex computational structures) and how deleterious race conditions can occur when dependencies among concurrent tasks are not respected. Also understand that there are situations where not respecting the order of certain computations may result in nondeterministic, but acceptable results. See examples under Mutual Exclusion and Conflict Resolution. Instructors may use this discussion to emphasize that distributed systems generally offer little control of time, and that executions are usually event-driven. For example, if a decision is to be made on the basis of emails from several sources, then the order of receipt of the emails should not affect the decision (as the order cannot be controlled).			CS1; CS2; DS/A

		● Task graphs	C	Show multiple examples of this concrete algorithmic abstraction as a mechanism for exposing inter-task dependencies. These graphs, which are used also in compiler analyses, form the level at which parallelism is exposed and exploited. Recognize that Series-parallel graphs are a special case of task graphs, which can emerge from barrier synchronizations or fork-join. Understand the possible penalties (in parallelism) that this synchronization incurs (Amdahl's law).			DS/A; SwEng
		● Makespan	K	Observe analyses in which makespan is identified with parallel time (basically, time to completion)			DS/A
		● Energy aware scheduling	N		C	Understand how processes can be scheduled to minimize energy consumption due, for example, by taking advantage of active and idle power management capabilities.	OS
Algorithmic Techniques				Learn a variety of techniques, patterns, or kernels whose algorithmic utility has been demonstrated over the course of decades.			
	Decomposition			Recognize that tasks and/or data associated with an algorithm need to be decomposed into parts to expose concurrency that can be exploited by computing elements running in parallel.			
		Recursion and Divide & Conquer (parallel aspects)	A	Recognize that the same structure that enables divide and conquer (sequential) algorithms exposes opportunities for parallel computation. Examples include mergesort or numerical integration (trapezoid rule, Simpson’s rule) or (at a more advanced level) Strassen's matrix-multiply.			CS2; DS/A; Algo2
		Blocking and Striping	N		C	See examples of this algorithmic manifestation of memory hierarchies	Algo2
		Architecture-Specific decomposition	N		C	Understand that performance and/or energy efficiency can be improved by decomposing an algorithm to exploit features of GPUs (as opposed to, for example, CPUs)	Algo2

	Load Balancing		K	Understand that processors equitably sharing the computational/communication load among processors benefits the entire algorithm. That is, the most stretched processor determines the performance of the entire algorithm. Use a simple task graph (for example) with a small number of processors to illustrate the idea.			DS/A; CS2
	Multi-party communication			Recognize the semantics of some common multi-party communication operations and how to use them appropriately in parallel algorithms.			
		Reduction	C	Recognize and use the tree structure implicit in applications such as scalar product, mergesort, histogram, mapreduce, etc.			DS/A
		Synchronization	A	Recognize that synchronization is necessary for certain algorithms to work correctly. Also recognize that synchronization should be used only when needed as it entails its own overheads. For example, a reduction tree implemented on multiple processing elements needs synchronization before the operation of an internal node is performed. However, a barrier synchronization across the entire tree at each level imposes an unnecessary logarithmic overhead; in this situation local synchronizations suffice.			CS1; CS2; DS/A
		Parallel Prefix (Scan)	K	Observe, via examples this "high-level" algorithmic tool. For instance, polynomial evaluation can be done as a prefix product (of powers of x), then a set of multiplications, followed by a reduction.			DS/A
		Other multi-party communication patterns	K	Recognize common multi-party communication patterns such as broadcast, gather, scatter, all-to-all or many-to-many communications, and their use as building blocks for parallel and distributed algorithms. Illustrate using block matrix transpose or shuffle from map-reduce, etc.			CS2; DS/A

MapReduce	N			Understand MapReduce as an approach to Big Data that involves data movement and then doing a reduction on like data. Understand a simple example such as word count. It is assumed that students know elements of a distributed model that emphasizes the infeasibility of moving large amounts of data.	Algo2
Mutual Exclusion and Conflict Resolution	C	Understand the need to resolve conflicts among concurrent processes competing for a shared resource. Here the computation may have to grant exclusive access to one process to ensure correctness and/or progress. Be able to identify and mitigate problems due to races. Instructors may provide examples such as (a) selecting an airline seat that may be simultaneously competed for by several passengers, (b) selecting which customer gets an item when more than one tries to buy it simultaneously, (c) mutual exclusion in the context of Java threads, (d) Dining philosophers.	A	This can also be explored in the context of MAC protocols (Networking); leader election (Distributed Systems). Asynchrony and local knowledge are causes of conflict (Distributed systems).	CS2; DS/A; Networking; DistSystems
Algorithmic Problems		The Algorithmic problems section contains parallel algorithms for certain problems. The important thing here is to emphasize the parallel/distributed aspects of the topic			
Algorithms for Communication and Synchronization		Understand (at the pseudo-code level) how certain patterns of communication can be implemented in a parallel/distributed model; the model(s) of choice (MoCs) could serve as good vehicle(s) on which to explore these ideas consistently across the course. As a corollary, one could also appreciate the cost of communication in PDC.			

		Reduction and Broadcast for communication and synchronization	C	Understand, for example, how recursive doubling can be used for all-to-one reduction, and its dual, one-to-all reduction, in log(p) steps. The same applies to all-to-all broadcast and all-to-all reduction. Be aware of the synonyms for these operations in the jargon associated with different areas; for example, all-to-all broadcast may be referred to as "gossip" or "total exchange". Recognize that all-to-all broadcast/reduction are synchronizing operations in a distributed (event-driven) environment.	A		DS/A; Algo2
		Parallel Prefix (Scan)	C	Understand the structure of at least one simple parallel prefix algorithm, for example, on a PRAM-type model. One could consider recursive or iterative approaches (such as those of Ladner-Fischer, Kogge-Stone, Brent-Kung)	A		DS/A; Algo2
		Multicast	N		C	Extend broadcast to multicast and explore avenues for communication-efficiency, relative to the MoC.	OS
		Permutation	N		C	Understand important permutations (shuffle, transpose etc.) and their implementation complexity issues.	OS
		Critical Regions and Mutual Exclusion	K	Be aware that a solution to the critical section problems must satisfy Mutual Exclusion, Progress, and Bounded Wait Times.	C		OS; DistSystems; ParProg
		Termination detection	K	Observe that, unlike the sequential case, processes in parallel and distributed algorithms may not know when the problem has been solved, or even if their part of the problem is solved; so termination has to be addressed explicitly. In some cases (such as reduction, tree algorithms, divide and conquer) it may be possible for a process to terminate on the basis of local information (for example, a node has passed its information to its parent in a reduction tree). In other cases, a global check may be necessary.	C	See examples that suggest the difficulty of proving that algorithms from various classes actually terminate. For more advanced courses, observe proofs of termination, to understand the conceptual tools needed.	DS/A; OS

	Sorting	K	Observe at least one parallel sorting algorithm together with analysis. Parallel merge sort is the simplest example, but other alternatives might be covered as well; more sophisticated algorithms might be covered in more advanced courses.	C		CS2; DS/A; Algo2
	Search	K	With the help of BFS- or DFS-like parallel search in a tree, graph or solution space, understand speedup anomalies and the fact that certain algorithms don't lend themselves to parallelization without modifying the semantics of the original problem. For example, strict DFS order of visiting nodes in a tree cannot be maintained in parallel. Also, the order in which solutions are found and the time taken to find them could vary unpredictably depending on the degree of parallelism. Detailed knowledge of parallel search algorithms is not expected.	C		DS/A; Algo2
	Algorithms for streams	K	Comprehend the notion of efficient algorithms (e.g., Bloom filters, heavy hitters) and structures (e.g., distributed hash tables) for stream data, and the difficulty of dealing with limited space.			CS1; DS/A
	Spatial Problems	N		K	Understand parallel and distributed decomposition, data structures, and solving strategies for problems rooted in a distribution of points in a multidimensional space such as n-body simulations (FMM, Barnes-Hut), data clustering (R* trees, DBSCAN), and classifiers (k-NN)	Algo2

	Deeper Algorithmic Experience	A	Experience through class instruction and assignment/project the design, analysis, and implementation aspects of at least one parallel or distributed algorithm of choice in detail. Master PDC algorithmic concepts through a detailed exploration, including recognizing how algorithm design reflects the structure of the computational problem(s) and the PDC model/environment. Possible computational problems to explore include matrix product, map reduce, sorting, search, convolution, a graph algorithm of your choice.			CS2; DS/A

8. Emerging Topics

This section includes topics of significant current or emerging interest that can serve as suitable backdrops to explore several PDC ideas; these topics include for instance, IoT, Collaborative computing and machine learning. Typically, these emerging topics will be explored in varying levels of depth in specific advanced courses such as AI or Advanced networking. In this section we point to directions in which these ideas can be explored in the PDC context. For example, parallel search trees and distributed databases can be cast in an AI backdrop. While many of these ideas are likely better suited for advanced courses, they may be introduced in lower level courses in a limited way. Additionally, many of these application topics will be familiar to students in their everyday lives and can thus serve as motivation for deeper inquiry. The set of topics in this category will necessarily be continually evolving as topics mature, retire, and newer topics emerge.

The current set of topics are organized into three broad groups: Emerging Distributed Systems, Distributed Applications, and Miscellaneous topics. Each of these groups has individual emerging areas listed. We have suggested learning outcomes for these group and individual topics. However recognizing the large variability in the manner in which these topics may be explored, we have used Bloom levels only for the groups.

Table 5: Emerging Topics

Emerging Topics		Core Bloom Level	Learning Outcome and Teaching Suggestions (core)	Advanced Bloom Level	Learning Outcome (advanced)	Where Covered
Emerging Distributed Systems		K	Know about different emerging distributed systems (for example some of those listed below).	C	Be able to identify new types of distributed systems (such as those listed below). Explain how they are similar to and differ from regular distributed systems; for example, one compare and contrast the Internet with an IoT system in a networking course.	Systems, DistSystems, Netw
	Internet of Things	N			Understand, for instance, that IoTs are resource constrained end point devices that are being used in many places such as smart homes, appliances etc. They are connected to the Internet often via a special network (eg modbus, bluetooth, DNP3)	Netw, DistSystems
	Edge Computing	N			Understand, for example, that in some distributed environment in-situ computing is necessary at the edge of the network where data is being generated or collected for reducing the communication cost and/or increasing response time.	DistSystems, Netw, Algo2
	Cyber Physical Systems	N			Understand that the cyber part, for instance the SCADA systems of cyber physical systems (eg smart grid) forms a distributed system. These distributed systems are heterogeneous in nature and often consist of a combination of purpose built as well as general purpose nodes and communication networks.	DistSystems, Netw
	Mobile, Adhoc and Software Defined Network	N			Understand that, for example, mobility and software defined networks admit a dynamic topology and a different relationship between communicating nodes. Explore the impact of these differences in the system.	DistSystems, ParProg, ParAlgo
Distributed Applications				C	Understand broader principles in the backdrop of specific applications; explore broader implications and considerations unique to certain applications	
	Social Networking	N			Understand that the rise of social networking provides new opportunities for enriching distributed computing with human & social context.	DistSystems, AI
	Web Services	N			Understand that web services provide functionalities to distributed agents. Explore web services in the context of client-server and	DistSystems, Netw, SwEng

					peer-to-peer models	
	Collaborative Computing	N			Understand that collaboration between multiple users (eg. zoom or Google Docs) or devices (example, robot swarms) is a form of distributed computing with application-specific requirements such as, synchronization, data consistency or action coordination.	DistSystems, ParAlgo, Netw
	Blockchain	N			Distributed computing is at the core of many blockchain related ideas. As an example, distributed consensus can be explored in the blockchain setting.	DistSystems, Security
Miscellaneous				K	Augment core PDC concepts in the context of emerging areas	
	Artificial Intelligence	N			Know that asynchrony, parallel search trees, and distributed databases form the basis for various branches of AI	ParProg, DB, AI, Machine Learning
	Machine Learning	N			Know that accelerators, parallel matrix decompositions, parallel matrix multiplication, and vectorized operations are fundamental aspects that enable modern machine learning	Systems, ParProg, Machine Learning
	Quantum Computing	N			Know that quantum computing algorithms can solve some problems (e.g., optimization problems) exponentially faster than classical algorithms via parallelism, and that quantum processors to implement some of those algorithms are under construction, using new parallel programming models	ParAlgo
	Security in Distributed Systems	N			Know that distributed systems are more vulnerable to privacy and security threats; distributed attacks modes; inherent tension between privacy and security.	DistSystems, Netw