Parallel and Distributed Computing (PD)

The past decade has brought explosive growth in multiprocessor computing, including multi-core processors and distributed data centers. As a result, parallel and distributed computing has moved from a largely elective topic to become more of a core component of undergraduate computing curricula. Both parallel and distributed computing entail the logically simultaneous execution of multiple processes, whose operations have the potential to interleave in complex ways. Parallel and distributed computing builds on foundations in many areas, including an understanding of fundamental systems concepts such as concurrency and parallel execution, consistency in state/memory manipulation, and latency. Communication and coordination among processes is rooted in the message-passing and shared-memory models of computing and such algorithmic concepts as atomicity, consensus, and conditional waiting. Achieving speedup in practice requires an understanding of parallel algorithms, strategies for problem decomposition, system architecture, detailed implementation strategies, and performance analysis and tuning. Distributed systems highlight the problems of security and fault tolerance, emphasize the maintenance of replicated state, and introduce additional issues that bridge to computer networking.

Because parallelism interacts with so many areas of computing, including at least algorithms, languages, systems, networking, and hardware, many curricula will put different parts of the knowledge area in different courses, rather than in a dedicated course. While we acknowledge that computer science is moving in this direction and may reach that point, in 2013 this process is still in flux and we feel it provides more useful guidance to curriculum designers to aggregate the fundamental parallelism topics in one place. Note, however, that the fundamentals of concurrency and mutual exclusion appear in the Systems Fundamentals (SF) Knowledge Area. Many curricula may choose to introduce parallelism and concurrency in the same course (see below for the distinction intended by these terms). Further, we note that the topics and learning outcomes listed below include only brief mentions of purely elective coverage. At the present time, there is too much diversity in topics that share little in common (including for example, parallel scientific computing, process calculi, and non-blocking data structures) to recommend particular topics be covered in elective courses.

Because the terminology of parallel and distributed computing varies among communities, we provide here brief descriptions of the intended senses of a few terms. This list is not exhaustive or definitive, but is provided for the sake of clarity.

- *Parallelism:* Using additional computational resources simultaneously, usually for speedup.
- *Concurrency:* Efficiently and correctly managing concurrent access to resources.
- *Activity*: A computation that may proceed concurrently with others; for example a program, process, thread, or active parallel hardware component.
- *Atomicity*: Rules and properties governing whether an action is observationally indivisible; for example, setting all of the bits in a word, transmitting a single packet, or completing a transaction.
- *Consensus*: Agreement among two or more activities about a given predicate; for example, the value of a counter, the owner of a lock, or the termination of a thread.
- *Consistency*: Rules and properties governing agreement about the values of variables written, or messages produced, by some activities and used by others (thus possibly exhibiting a *data race*); for example, *sequential consistency*, stating that the values of all variables in a shared memory parallel program are equivalent to that of a single program performing some interleaving of the memory accesses of these activities.
- *Multicast*: A message sent to possibly many recipients, generally without any constraints about whether some recipients receive the message before others. An *event* is a multicast message sent to a designated set of *listeners* or *subscribers*.

As multi-processor computing continues to grow in the coming years, so too will the role of parallel and distributed computing in undergraduate computing curricula. In addition to the guidelines presented here, we also direct the interested reader to the document entitled "NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates", available from the website: http://www.cs.gsu.edu/~tcpp/curriculum/.

General cross-referencing note: Systems Fundamentals also contains an introduction to parallelism (SF/Computational Paradigms, SF/System Support for Parallelism, SF/Performance).

The introduction to parallelism in SF complements the one here and there is no ordering constraint between them. In SF, the idea is to provide a unified view of the system support for simultaneous execution at multiple levels of abstraction (parallelism is inherent in gates, processors, operating systems, and servers), whereas here the focus is on a preliminary understanding of parallelism as a computing primitive and the complications that arise in parallel and concurrent programming. Given these different perspectives, the hours assigned to each are not redundant: the layered systems view and the high-level computing concepts are accounted for separately in terms of the core hours.

PD. Parallel and Distributed Computing (5 Core-Tier1 hours, 10 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PD/Parallelism Fundamentals	2		N
PD/Parallel Decomposition	1	3	N
PD/Communication and Coordination	1	3	Y
PD/Parallel Algorithms, Analysis, and Programming		3	Y
PD/Parallel Architecture	1	1	Y
PD/Parallel Performance			Y
PD/Distributed Systems			Y
PD/Cloud Computing			Y
PD/Formal Models and Semantics			Y

PD/Parallelism Fundamentals

[2 Core-Tier1 hours]

Build upon students' familiarity with the notion of basic parallel execution—a concept addressed in Systems Fundamentals—to delve into the complicating issues that stem from this notion, such as race conditions and liveness.

Cross-reference SF/Computational Paradigms and SF/System Support for Parallelism.

Topics:

- Multiple simultaneous computations
- Goals of parallelism (e.g., throughput) versus concurrency (e.g., controlling access to shared resources)
- Parallelism, communication, and coordination
    - Programming constructs for coordinating multiple simultaneous computations
    - Need for synchronization
- Programming errors not found in sequential programming
    - Data races (simultaneous read/write or write/write of shared state)
    - Higher-level races (interleavings violating program intention, undesired non-determinism)
    - Lack of liveness/progress (deadlock, starvation)

Learning outcomes:

1. Distinguish using computational resources for a faster answer from managing efficient access to a shared resource. (Cross-reference GV/Fundamental Concepts, outcome 5.) [Familiarity]
2. Distinguish multiple sufficient programming constructs for synchronization that may be inter-implementable but have complementary advantages. [Familiarity]
3. Distinguish data races from higher level races. [Familiarity]

PD/Parallel Decomposition

[1 Core-Tier1 hour, 3 Core-Tier2 hours]

(Cross-reference SF/System Support for Parallelism)

Topics:

[Core-Tier1]

- Need for communication and coordination/synchronization
- Independence and partitioning

[Core-Tier2]

- Basic knowledge of parallel decomposition concepts (cross-reference SF/System Support for Parallelism)
- Task-based decomposition
    - Implementation strategies such as threads
- Data-parallel decomposition
    - Strategies such as SIMD and MapReduce
- Actors and reactive processes (e.g., request handlers)

Learning outcomes:

[Core-Tier1]

1. Explain why synchronization is necessary in a specific parallel program. [Usage]
2. Identify opportunities to partition a serial program into independent parallel modules. [Familiarity]

[Core-Tier2]

3. Write a correct and scalable parallel algorithm. [Usage]
4. Parallelize an algorithm by applying task-based decomposition. [Usage]
5. Parallelize an algorithm by applying data-parallel decomposition. [Usage]
6. Write a program using actors and/or reactive processes. [Usage]

PD/Communication and Coordination

[1 Core-Tier1 hour, 3 Core-Tier2 hours]

Cross-reference OS/Concurrency for mechanism implementation issues.

Topics:

[Core-Tier1]

- Shared Memory
- Consistency, and its role in programming language guarantees for data-race-free programs

[Core-Tier2]

- Message passing
  - Point-to-point versus multicast (or event-based) messages
  - Blocking versus non-blocking styles for sending and receiving messages
  - Message buffering (cross-reference PF/Fundamental Data Structures/Queues)
- Atomicity
  - Specifying and testing atomicity and safety requirements
  - Granularity of atomic accesses and updates, and the use of constructs such as critical sections or transactions to describe them
  - Mutual Exclusion using locks, semaphores, monitors, or related constructs
    - Potential for liveness failures and deadlock (causes, conditions, prevention)
  - Composition
    - Composing larger granularity atomic actions using synchronization
    - Transactions, including optimistic and conservative approaches

[Elective]

- Consensus
  - (Cyclic) barriers, counters, or related constructs
- Conditional actions
  - Conditional waiting (e.g., using condition variables)

Learning outcomes:

[Core-Tier1]

1. Use mutual exclusion to avoid a given race condition. [Usage]
2. Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent. [Familiarity]

[Core-Tier2]

3. Give an example of a scenario in which blocking message sends can deadlock. [Usage]
4. Explain when and why multicast or event-based messaging can be preferable to alternatives. [Familiarity]
5. Write a program that correctly terminates when all of a set of concurrent tasks have completed. [Usage]
6. Use a properly synchronized queue to buffer data passed among activities. [Usage]
7. Explain why checks for preconditions, and actions based on these checks, must share the same unit of atomicity to be effective. [Familiarity]
8. Write a test program that can reveal a concurrent programming error; for example, missing an update when two activities both try to increment a variable. [Usage]
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks or semaphores. [Familiarity]
10. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates. [Familiarity]
11. Give an example of a scenario in which an attempted optimistic update may never complete. [Familiarity]

[Elective]

12. Use semaphores or condition variables to block threads until a necessary precondition holds. [Usage]

PD/Parallel Algorithms, Analysis, and Programming

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Critical paths, work and span, and the relation to Amdahl's law (cross-reference SF/Performance)
- Speed-up and scalability
- Naturally (embarrassingly) parallel algorithms
- Parallel algorithmic patterns (divide-and-conquer, map and reduce, master-workers, others)
    - Specific algorithms (e.g., parallel MergeSort)

[Elective]

- Parallel graph algorithms (e.g., parallel shortest path, parallel spanning tree) (cross-reference AL/Algorithmic Strategies/Divide-and-conquer)
- Parallel matrix computations
- Producer-consumer and pipelined algorithms
- Examples of non-scalable parallel algorithms

Learning outcomes:

[Core-Tier2]

1. Define "critical path", "work", and "span". [Familiarity]
2. Compute the work and span, and determine the critical path with respect to a parallel execution diagram. [Usage]
3. Define "speed-up" and explain the notion of an algorithm's scalability in this regard. [Familiarity]
4. Identify independent tasks in a program that may be parallelized. [Usage]
5. Characterize features of a workload that allow or prevent it from being naturally parallelized. [Familiarity]
6. Implement a parallel divide-and-conquer (and/or graph algorithm) and empirically measure its performance relative to its sequential analog. [Usage]
7. Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations. [Usage]

[Elective]

8. Provide an example of a problem that fits the producer-consumer paradigm. [Familiarity]
9. Give examples of problems where pipelining would be an effective means of parallelization. [Familiarity]
10. Implement a parallel matrix algorithm. [Usage]
11. Identify issues that arise in producer-consumer algorithms and mechanisms that may be used for addressing them. [Familiarity]

PD/Parallel Architecture

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

The topics listed here are related to knowledge units in the Architecture and Organization (AR) knowledge area (AR/Assembly Level Machine Organization and AR/Multiprocessing and Alternative Architectures). Here, we focus on parallel architecture from the standpoint of applications, whereas the Architecture and Organization knowledge area presents the topic from the hardware perspective.

[Core-Tier1]

- Multicore processors
- Shared vs. distributed memory

[Core-Tier2]

- Symmetric multiprocessing (SMP)
- SIMD, vector processing

[Elective]

- GPU, co-processing
- Flynn's taxonomy
- Instruction level support for parallel programming
    - Atomic instructions such as Compare and Set
- Memory issues
    - Multiprocessor caches and cache coherence
    - Non-uniform memory access (NUMA)

- Topologies
    - Interconnects
    - Clusters
    - Resource sharing (e.g., buses and interconnects)

Learning outcomes:

[Core-Tier1]

1. Explain the differences between shared and distributed memory. [Familiarity]

[Core-Tier2]

2. Describe the SMP architecture and note its key features. [Familiarity]
3. Characterize the kinds of tasks that are a natural match for SIMD machines. [Familiarity]

[Elective]

4. Describe the advantages and limitations of GPUs vs. CPUs. [Familiarity]
5. Explain the features of each classification in Flynn's taxonomy. [Familiarity]
6. Describe assembly-level support for atomic operations. [Familiarity]
7. Describe the challenges in maintaining cache coherence. [Familiarity]
8. Describe the key performance challenges in different memory and distributed system topologies. [Familiarity]

PD/Parallel Performance

[Elective]

Topics:

- Load balancing
- Performance measurement
- Scheduling and contention (cross-reference OS/Scheduling and Dispatch)
- Evaluating communication overhead
- Data management
    - Non-uniform communication costs due to proximity (cross-reference SF/Proximity)
    - Cache effects (e.g., false sharing)
    - Maintaining spatial locality
- Power usage and management

Learning outcomes:

1. Detect and correct a load imbalance. [Usage]
2. Calculate the implications of Amdahl's law for a particular parallel algorithm (cross-reference SF/Evaluation for Amdahl's Law). [Usage]
3. Describe how data distribution/layout can affect an algorithm's communication costs. [Familiarity]
4. Detect and correct an instance of false sharing. [Usage]
5. Explain the impact of scheduling on parallel performance. [Familiarity]
6. Explain performance impacts of data locality. [Familiarity]
7. Explain the impact and trade-off related to power usage on parallel performance. [Familiarity]

PD/Distributed Systems

[Elective]

Topics:

- Faults (cross-reference OS/Fault Tolerance)
    - Network-based (including partitions) and node-based failures
    - Impact on system-wide guarantees (e.g., availability)
- Distributed message sending
    - Data conversion and transmission
    - Sockets
    - Message sequencing
    - Buffering, retrying, and dropping messages
- Distributed system design tradeoffs
    - Latency versus throughput
    - Consistency, availability, partition tolerance
- Distributed service design
    - Stateful versus stateless protocols and services
    - Session (connection-based) designs
    - Reactive (IO-triggered) and multithreaded designs
- Core distributed algorithms
    - Election, discovery

Learning outcomes:

1. Distinguish network faults from other kinds of failures. [Familiarity]
2. Explain why synchronization constructs such as simple locks are not useful in the presence of distributed faults. [Familiarity]
3. Write a program that performs any required marshaling and conversion into message units, such as packets, to communicate interesting data between two hosts. [Usage]
4. Measure the observed throughput and response latency across hosts in a given network. [Usage]
5. Explain why no distributed system can be simultaneously consistent, available, and partition tolerant. [Familiarity]
6. Implement a simple server -- for example, a spell checking service. [Usage]
7. Explain the tradeoffs among overhead, scalability, and fault tolerance when choosing a stateful v. stateless design for a given service. [Familiarity]
8. Describe the scalability challenges associated with a service growing to accommodate many clients, as well as those associated with a service only transiently having many clients. [Familiarity]
9. Give examples of problems for which consensus algorithms such as leader election are required. [Usage]

PD/Cloud Computing

[Elective]

Topics:

- Internet-Scale computing
    - Task partitioning (cross-reference PD/Parallel Algorithms, Analysis, and Programming)
    - Data access
    - Clusters, grids, and meshes
- Cloud services
    - Infrastructure as a service
        - Elasticity of resources
        - Platform APIs

    - Software as a service
    - Security
    - Cost management
- Virtualization (cross-reference SF/Virtualization and Isolation and OS/Virtual Machines)
    - Shared resource management
    - Migration of processes
- Cloud-based data storage
    - Shared access to weakly consistent data stores
    - Data synchronization
    - Data partitioning
    - Distributed file systems (cross-reference IM/Distributed Databases)
    - Replication

Learning outcomes:

1. Discuss the importance of elasticity and resource management in cloud computing. [Familiarity]
2. Explain strategies to synchronize a common view of shared data across a collection of devices. [Familiarity]
3. Explain the advantages and disadvantages of using virtualized infrastructure. [Familiarity]
4. Deploy an application that uses cloud infrastructure for computing and/or data resources. [Usage]
5. Appropriately partition an application between a client and resources. [Usage]

PD/Formal Models and Semantics

[Elective]

Topics:

- Formal models of processes and message passing, including algebras such as Communicating Sequential Processes (CSP) and pi-calculus
- Formal models of parallel computation, including the Parallel Random Access Machine (PRAM) and alternatives such as Bulk Synchronous Parallel (BSP)
- Formal models of computational dependencies
- Models of (relaxed) shared memory consistency and their relation to programming language specifications
- Algorithmic correctness criteria including linearizability
- Models of algorithmic progress, including non-blocking guarantees and fairness
- Techniques for specifying and checking correctness properties such as atomicity and freedom from data races

Learning outcomes:

1. Model a concurrent process using a formal model, such as pi-calculus. [Usage]
2. Explain the characteristics of a particular formal parallel model. [Familiarity]
3. Formally model a shared memory system to show if it is consistent. [Usage]
4. Use a model to show progress guarantees in a parallel algorithm. [Usage]
5. Use formal techniques to show that a parallel algorithm is correct with respect to a safety or liveness property. [Usage]
6. Decide if a specific execution is linearizable or not. [Usage]