Parallel and Distributed Computing (PDC)

Preamble

Parallel and distributed programming arranges and controls multiple computations occurring at the same time across different places. The ubiquity of parallelism and distribution are inevitable consequences of increasing numbers of gates in processors, processors in computers, and computers everywhere that may be used to improve performance compared to sequential programs, while also coping with the intrinsic interconnectedness of the world, and the possibility that some components or connections fail or misbehave. Parallel and distributed programming remove the restrictions of sequential programming that require computational steps to occur in a serial order in a single place, revealing further distinctions, techniques, and analyses applying at each layer of computing systems.

In most conventional usage, “parallel” programming focuses on arranging that multiple activities co-occur, “distributed” programming focuses on arranging that activities occur in different places, and “concurrent” programming focuses on interactions of ongoing activities with each other and the environment. However, all three terms may apply in most contexts. Parallelism generally implies some form of distribution because multiple activities occurring without sequential ordering constraints happen in multiple physical places (unless relying on context-switching schedulers or quantum effects). And conversely, actions in different places need not bear any particular sequential ordering with respect to each other in the absence of communication constraints. The focus of this KA is on contexts in which parallelism and distribution are explicitly introduced, but also apply when they are required by other constraints, such as when required data or services are intrinsically remote.

It can be challenging to teach and learn about an area that has evolved from being a diverse set of advanced topics into a central body of knowledge and practice, permeating almost every other aspect of computing. Nearly every problem with a sequential solution also admits parallel and/or distributed solutions; additional problems and solutions arise only in the context of existing concurrency. And nearly every application domain of parallel and distributed computing is a well-developed area of study and/or engineering too large to enumerate. Growth of the field has occurred irregularly across different subfields of computing, sometimes with different goals, terminology, practices, and associated courses, and sometimes masking the considerable overlap of basic ideas and skills that are the main focus of this KA.

Changes since CS 2013:

<table>
<thead>
<tr>
<th>Knowledge Units</th>
<th>CS Core hours</th>
<th>KA Core hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs and Execution</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
CS Core topics span approaches to and aspects of parallel and distributed computing, but restrict coverage to those applying to nearly all of them. The main focus is on removing limitations of strictly sequential programming, revealing the essential structure and properties of parallel and distributed systems and software. Learning Outcomes include developing small programs (in a choice of several styles) with multiple activities and analyzing basic properties. The topics and hours do not include coverage of particular languages, tools, frameworks, systems, and platforms that would normally be included in any given course as a basis for implementing and evaluating concepts and skills. They also avoid reliance on specific choices that may vary widely (for example GPU programming vs cloud container deployment scripts), and include only brief mentions of related content more closely associated with Programming Languages, Computer Architecture, Networking, Security, and Systems KAs.

KA Core topics in each unit are of the form “One or more of the following” for a set of topics that extend associated core topics, These permit variation in coverage depending on the focus of any given course. As discussed further below, examples include a High Performance Computing course focusing on heterogeneous data parallelism mainly applied to linear algebra problems, a systems-oriented concurrency course focusing on shared-memory coordination mainly applied to resource management and middleware support, a distributed data processing course focusing on fault-tolerant data stores applied to web commerce applications, and so on. Depth of coverage of any KA Core subtopic is expected to vary according to course goals. For example, shared-memory coordination is a central topic in multicore programming, but much less so in most heterogeneous systems, and conversely for bulk data transfer. Similarly, fault tolerance is central to the design of distributed software, but much less so in most data-parallel applications.

### Knowledge Units

<table>
<thead>
<tr>
<th>PD/Programs and Execution [2 CS Core hours, 2 KA Core hours]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topics</strong></td>
</tr>
<tr>
<td>○ [CS Core] Definitions and properties</td>
</tr>
<tr>
<td>■ Parallelizable actions</td>
</tr>
<tr>
<td>● Including closures, functions, and services</td>
</tr>
<tr>
<td>● Composite actions; sessions, tasks; scopes</td>
</tr>
<tr>
<td>● Naming or identifying actions as parties (for example thread IDs)</td>
</tr>
</tbody>
</table>
Impact of granularity and overhead

Ordering among actions; happens-before relations
- Relaxing ordering constraints permits nondeterministic execution of the series/parallel directed acyclic graphs representing program components

Independence: determining when ordering doesn’t matter, in terms of commutativity, dependencies, preconditions

Consistency: Agreement about values and predicates; races, atomicity, consensus

Ensuring ordering when necessary in parallel programs
- For example locking, safe publication
- Communication among parties may impose ordering among their actions. As in: sending a message happens before receiving it

Places: Physical devices executing parallel actions (parties)
- Hardware components, remote hosts
- Places may support multiple parties (in sequence or schedule)
- Parties may include external and human users
- May include scheduling, time-slicing and emulation of multiple parallel actions by fewer processors

Faults arising from failures in parties or communication
- Failures may be due to untrusted parties and protocols not under the control of the program or administrative domain
- Degree of fault tolerance may be a design choice

[CS Core] Starting parallel actions
- Placement: arranging that the action be performed (eventually) by a designated party
  - Details range from from hardwiring to configuration scripts, or relying on automated provisioning and management by platforms
  - Establishing communication and resource management
- Procedural: Enabling multiple actions to start at a given program point
  - For example, starting new threads
  - May be contained in a scope ending when all complete
- Reactive: Enabling upon an event
  - For example, installing event handlers
  - Less control of when actions begin or end
- Dependent: Enabling upon completion of others
  - For example, sequencing

[KA Core] Underlying mappings and mechanisms. **One or more of:**
- CPU data- and instruction-level- parallelism
- SIMD and heterogeneous data parallelism
- Multicore scheduled concurrency, tasks, actors
- Clusters, clouds; elastic provisioning
- Distributed systems with unbounded participants
Emerging technologies such as quantum computing and molecular computing

Illustrative Learning Outcomes

- [CS Core] Graphically show (as a dag) how to parallelize a compound numerical expression; for example $a = (b+c) \times (d + e)$.
- [CS Core] Identify a race error in a given program
- [CS Core] In a given context, explain the extent to which introducing parallelism in an otherwise sequential program would be expected to improve throughput and/or reduce latency, and how it may impact energy efficiency
- [KA-Core] Write a function that efficiently counts events such as networking packet receptions
- [KA-Core] Write a filter/map/reduce program in multiple styles
- [KA-Core] Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PD/Communication [2 CS core hours, 6 KA Core hours]

Topics

- [CS Core] Fundamentals
  - Media
    - Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores
    - Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation
  - Channels
    - Explicit party-to-party communication; naming
    - APIs: sockets, architectural and language-based channel constructs
  - Memory
    - Architectures in which parties directly communicate only with memory at given addresses
    - Consistency: Bitwise atomicity limits, coherence, local ordering
    - Memory hierarchies, locality: caches, latency, false-sharing
    - Heterogeneous Memory using multiple memory stores, with explicit data transfer across them; for example, GPU local and shared memory, DMA
    - Multiple layers of sharing domains, scopes and caches
  - Data Stores
    - Cooperatively maintained structured data implementing maps, sets, and related ADTs
    - Varieties: Owned, shared, sharded, replicated, immutable, versioned
- [CS Core] Programming with communication
  - Using channel, socket, and/or remote procedure call APIs
  - Using shared memory constructs in a given language
[KA Core] Properties and Extensions. **One or more of:**

- **Media**
  - Topologies: Unicast, Multicast, Mailboxes, Switches; Routing
  - Concurrency properties: Ordering, consistency, idempotency, overlapping with computation
  - Reliability: transmission errors and drops.
  - Data formats, marshaling
  - Protocol design: progress guarantees, deadlocks
  - Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion), Responsiveness (liveness).
  - Applications of Queuing Theory to model and predict performance

- **Channels**
  - Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control
  - Program control for sending (usually procedural) vs receiving (usually reactive or RPC-based)
  - Formats, marshaling, validation, encryption, compression
  - Multiplexing and demultiplexing in contexts with many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs.
  - Formalization and analysis; for example using CSP

- **Memory**
  - Memory models: sequential and release/acquire consistency
  - Memory management; including reclamation of shared data; reference counts and alternatives
  - Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays
  - Emulating shared memory: distributed shared memory, RDMA

- **Data Stores**
  - Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains,
  - Faults and partial failures; voting; protocols such as Paxos and Raft
  - Security and trust: Byzantine failures, proof of work and alternatives

- **Illustrative Learning Outcomes**
  - **[CS Core]** Determine whether shared memory or message passing would be preferable for a given application in a given context
  - **[CS Core]** Write a producer-consumer program in which one component generates numbers, and another computes their average. Measure speedups when the numbers are small scalars versus large multi-precision values.
○ [KA-Core] Write a program that distributes different segments of a data set to multiple workers, and collects results (for the simplest example, summing segments of an array).
○ [KA-Core] Write a parallel program that requests data from multiple sites, and summarizes them using some form of reduction
○ [KA-Core] Compare the performance of buffered versus unbuffered versions of a producer-consumer program
○ [KA-Core] Determine whether a given communication scheme provides sufficient security properties for a given usage
○ [KA-Core] Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
○ [KA-Core] Give an example of a scenario in which blocking message sends can deadlock.
○ [KA-Core] Describe at least one design technique for avoiding liveness failures in programs using multiple locks
○ [KA-Core] Write a program that illustrates memory-access or message reordering.
○ [KA-Core] Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
○ [KA-Core] Give an example of a scenario in which an attempted optimistic update may never complete.
○ [KA-Core] Modify a concurrent system to use a more scalable, reliable or available data store

PD/Coordination [2 CS core hours, 6 KA Core hours]

● Topics
  ○ [CS Core] Fundamentals
    ■ Dependent actions
      ● Execution control when one activity’s initiation or progress depends on actions of others
      ● Completion-based: Barriers, joins
      ● Data-enabled: Produce-Consumer designs
      ● Condition-based: Polling, retrying, backoffs, helping, suspension, queueing, signaling, timeouts
      ● Reactive: enabling and triggering continuations
    ■ Progress
      ● Dependency cycles and deadlock; monotonicity of conditions
    ■ Atomicity
      ● Atomic instructions, enforced local access orderings
      ● Locks and mutual exclusion
      ● Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers
      ● Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations, deadlock
  ○ [CS Core] Programming with coordination
    ■ Controlling termination
- Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races
- Using transactional APIs in a given framework
  - [KA Core] Properties and extensions. **One or more of:**
    - **Progress**
      - Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability
      - Performance: contention, granularity, convoying, scaling
      - Non-blocking data structures and algorithms
    - **Atomicity**
      - Ownership and resource control
      - Lock variants: sequence locks, read-write locks; reentrancy; tickets
      - Transaction-based control: Optimistic and conservative
      - Distributed locking: reliability
    - Interaction with other forms of program control
      - Alternatives to barriers: Clocks; Counters, virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors
      - Speculation, selection, cancellation; observability and security consequences
      - Resource-based: Semaphores and condition variables
      - Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.
      - Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting
  - **Illustrative Learning Outcomes**
    - [CS Core] Show how to avoid or repair a race error in a given program
    - [CS Core] Write a program that correctly terminates when all of a set of concurrent tasks have completed.
    - [KA-Core] Write a function that efficiently counts events such as sensor inputs or networking packet receptions
    - [KA-Core] Write a filter/map/reduce program in multiple styles
    - [KA-Core] Write a program in which the termination of one set of parallel actions is followed by another
    - [KA-Core] Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
    - [KA-Core] Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others
    - [KA-Core] Write a program for multiple parties to agree upon the current time of day; discuss its limitations compared to protocols such as NTP
    - [KA-Core] Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client
PD/Software Engineering [2 CS Core hours, 3 KA Core hours]

- **Topics**
  - [CS Core] Safety, liveness and performance requirements
    - Temporal logic constructs to express “always” and “eventually”
    - Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements
  - [CS Core] Identifying, testing for, and repairing violations
    - Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock)
  - [KA Core] Specification and Evaluation. **One or more of:**
    - **Formal Specification**
      - Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications
      - Use of tools such as UML, TLA, program logics
      - Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting
    - **Static Analysis**
      - For correctness, throughput, latency, resources, energy
      - dag model analysis of algorithmic efficiency (work, span, critical paths)
    - **Empirical Evaluation**
      - Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, continuous integration, continuous deployment, and test generators
      - Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads.
  - **Application domain specific analyses and evaluation techniques**

- **Illustrative Learning Outcomes**
  - [CS Core] Revise a specification to enable parallelism and distribution without violating other essential properties or features
  - [CS Core] Explain why avoiding non-local side-effects is a common design goal
  - Specify a set of invariants that must hold at each bulk-parallel step of a computation
  - [CS Core] Write a test program that can reveal a data race error; for example, missing an update when two activities both try to increment a variable.
  - [KA-Core] Specify and measure behavior when a service is requested by too many clients
○ [KA-Core] Identify and repair a performance problem due to sequential bottlenecks
○ [KA-Core] Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
○ [KA Core] Identify and repair a performance problem due to communication or data latency
○ [KA-Core] Identify and repair a performance problem due to communication or data latency
○ [KA-Core] Identify and repair a performance problem due to resource management overhead
○ [KA-Core] Identify and repair a reliability or availability problem

PD/Algorithms and Application Domains 2 CS Core hours, 9 KA Core hours]

● Topics
  ○ [CS Core] Expressing and implementing parallel and distributed algorithms
    ■ Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs
  ○ [CS Core] Survey:

<table>
<thead>
<tr>
<th>Category</th>
<th>Typical Execution agents</th>
<th>Typical Communication mechanisms</th>
<th>Typical Algorithmic domains</th>
<th>Typical Engineering goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore</td>
<td>threads</td>
<td>Shared memory, Atomics, locks</td>
<td>Resource management, data processing</td>
<td>throughput, latency, energy</td>
</tr>
<tr>
<td>Reactive</td>
<td>Handlers, threads</td>
<td>IO Channels</td>
<td>Services, real-time</td>
<td>latency</td>
</tr>
<tr>
<td>Data parallel</td>
<td>GPU, SIMD, accelerators, hybrid</td>
<td>Heterogeneous memory</td>
<td>Linear algebra, graphics, data analysis</td>
<td>throughput, energy</td>
</tr>
<tr>
<td>Cluster</td>
<td>Managed hosts</td>
<td>Sockets, Message channels</td>
<td>Simulation, data analysis</td>
<td>throughput</td>
</tr>
<tr>
<td>Cloud</td>
<td>Provisioned hosts</td>
<td>Service APIs</td>
<td>Web applications</td>
<td>scalability</td>
</tr>
<tr>
<td>Open Distributed</td>
<td>Autonomous hosts</td>
<td>Sockets, Data stores</td>
<td>Fault tolerant data stores and services</td>
<td>reliability</td>
</tr>
</tbody>
</table>

○ [KA Core] Algorithmic Domains. **One of more of:**
- Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning
- Data processing: sorting, searching and retrieval, concurrent data structures
- Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics
- Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms
- Logic: SAT, concurrent logic programming
- Graphics and computational geometry: Transforms, rendering, ray-tracing
- Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms, Batching, prioritization, partitioning, decentralization via work-stealing and related techniques
- Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.

- Illustrative Learning Outcomes
  - [CS Core] Implement a parallel/distributed component based on a known algorithm
  - [CS Core] Write a data-parallel program that for example computes the average of an array of numbers.
  - [CS Core] Extend an event-driven sequential program by establishing a new activity in an event handler (for example a new thread in a GUI action handler).
  - [CS Core] Improve the performance of a sequential component by introducing parallelism and/or distribution
  - [CS Core] Choose among different parallel/distributed designs for components of a given system
  - [KA-Core] Design, implement, analyze, and evaluate a component or application for X operating in a given context, where X is in one of the listed domains; for example a genetic algorithm for factory floor design.
  - [KA-Core] Critique the design and implementation of an existing component or application, or one developed by classmates
  - [KA-Core] Compare the performance and energy efficiency of multiple implementations of a similar design; for example multicore versus clustered versus GPU.

Professional Dispositions

Math Requirements
Shared Concepts and Crosscutting Themes

Shared Concepts:

 Dependencies. Prerequisites for the CS core of the PD Knowledge Area include familiarity with:

- SDF (Software Development Fundamentals): programs vs executions, specifications vs implementations, variables, sequential control flow (conditionals, loops), procedure/function/method calls; arrays.
- MF (Math Fundamentals): logic, discrete structures including directed graphs, state machines. Plus at least minimal familiarity of linear algebra, differential equations
- PL (Programming languages) Event-driven, OO, functional styles and constructs
- SF (System Fundamentals): layered systems, Von Neumann architecture.

Prerequisites for KA core topics vary across options

Overlaps. Some of the following coverage could be placed in courses primarily focused on other KAs:

- SF (System Fundamentals): RPC, performance evaluation
- NC (Networking and communication): Protocols, APIs
- OS (Operating Systems) Concurrency, scheduling, fault tolerance
- PL (Programming languages) Interactions of parallelism with other forms of program control and semantics
- SE (Software Engineering) Requirements and analysis
- SEC (Security) TBD

There is also potential overlap with most other KAs, depending on coverage choices in Algorithms and Applications

Course Packaging Suggestions

The modest PD CS Core topic requirements do not constitute a specific course. They may be incorporated by extending coverage in one or more courses primarily devoted to one more of the following KAs:

- Programming Language (KU PL/PD)
- System Fundamentals (KU SF/xx)
- Operating Systems (KU OS/xx)
- Data Management (KU DM/xx)
- Networking (KU NC/xx)
- Computer Architecture (KU AR/xx)
- Algorithms (KU AL/xx)

Alternatively, or in addition, the PD CS core may serve as a basis for courses focusing on parallel and/or distributed computing. At one extreme, it is possible to offer a single broadly
constructed course covering all PD KA Core topics to varying depths. At the other extreme, it is possible to infuse PD KA Core coverage across the curriculum by uniformly providing courses that cover parallel and distributed approaches alongside sequential ones for nearly every topic in computing. More conventional choices include courses that focus on one or a few categories (such as multicore or cluster), and algorithmic domains (such as linear algebra, or resource management). Such courses may go into further depth than listed in one or more KU, including development experience, but include only CS-Core-level coverage of other topics.

For the sake of a concrete example, here is a sample set of choices for a course mainly focusing on multicores:

1. Programs and Execution: CS Core plus KA Core on threads, tasks, instruction-level parallelism
2. Communication: CS Core plus KA core on multicore architectures, memory, concurrent data stores
3. Coordination: CS Core plus KA core on blocking and non-blocking synchronization, speculation, cancellation, futures, and divide-and-conquer data parallelism
4. Software Engineering: CS Core plus KA core on performance analysis
5. Algorithms and Applications: CS Core plus project-based KA Core coverage of data processing and resource management.

Competency Specifications

- **Task 1:** Implement a parallel/distributed component based on a known algorithm
  - **Competency Statement:** Implement parallel and distributed computing concepts that may be expressed in different ways across different languages and frameworks;
  - **Competency area:** Software
  - **Competency unit:** Development
  - **Required knowledge areas and knowledge units:**
    - PDC / Algorithms and Applications
    - PL / Parallel and Distributed Computing
  - **Required skill level:** Apply/Develop
  - **Core level:** CS core

- **Task 2:** Improve the performance of a sequential application or component by introducing parallelism and/or distribution
  - **Competency Statement:** Evaluate how and when parallelism and/or distribution can improve (or not improve) performance well enough to identify opportunities, as well as implement them and measure results
- **Competency area:** Software / Systems / Application
- **Competency unit:** Design/Development//Evaluation///Improvement
- **Required knowledge areas and knowledge units:**
  - PDC / Software Engineering
  - PL / Parallel and Distributed Computing
- **Required skill level:** Evaluate/Develop
- **Core level:** CS core

- **Task 3:** Revise a specification to enable parallelism and distribution without violating other essential properties or features
  - **Competency Statement:** Ensure that relaxing sequential constraints and/or remotely communicating do not have unexpected consequences that break existing software.
  - **Competency area:** Software / Systems / Application
  - **Competency unit:** Design/Maintenance
  - **Required knowledge areas and knowledge units:**
    - PDC / Software Engineering
    - PL / Parallel and Distributed Computing
  - **Required skill level:** Explain/Evaluate
  - **Core level:** CS core

- **Task 4:** Choose among different parallel/distributed designs for components of a given system
  - **Competency Statement:** Evaluate the relative merits of, for example, data parallel versus reactive designs are most applicable to problems at hand.
  - **Competency area:** Software / Systems / Application
  - **Competency unit:** Design/Evaluation
  - **Required knowledge areas and knowledge units:**
    - PDC / Algorithms and Applications
    - PL / Parallel and Distributed Computing
  - **Required skill level:** Evaluate
  - **Core level:** CS core

---

**Committee**

**Chair:** Doug Lea, State University of New York at Oswego, Oswego, USA

**Members:**
- Sherif Aly, American University of Cairo, Cairo, Egypt
## Appendix: Core Topics and Skill Levels

<table>
<thead>
<tr>
<th>KA</th>
<th>KU</th>
<th>Topic</th>
<th>Skill</th>
<th>Core</th>
<th>Hours</th>
</tr>
</thead>
</table>
| PD | Programs and Executions | Definitions and Properties  
   - Parallelizable actions  
   - Ordering among actions; happens-before relations  
   - Independence: determining when ordering doesn’t matter in terms of commutativity, dependencies, preconditions  
   - Consistency: Agreement about values and predicates; races, atomicity, consensus  
   - Ensuring ordering when necessary in parallel programs  
   - Places: Physical devices executing parallel actions (parties)  
   - Faults arising from failures in parties or communication | Explain | CS | 1 |
| PD | Programs and Executions | Starting parallel actions  
   - Placement: arranging that the action be performed (eventually) by a designated party  
   - Procedural: Enabling multiple actions to start at a given program point  
   - Reactive: Enabling upon an event  
   - Dependent: Enabling upon completion of others | Explain | CS | 1 |
| PD          | Programs and Executions | Underlying mappings and mechanisms. **One or more of:**  
|            |                          | - CPU data- and instruction-level-parallelism  
|            |                          | - SIMD and heterogeneous data parallelism  
|            |                          | - Multicore scheduled concurrency, tasks, actors  
|            |                          | - Clusters, clouds; elastic provisioning  
|            |                          | - Distributed systems with unbounded participants  
|            |                          | - Emerging technologies such as quantum computing and molecular computing  
|            |                          | **Communications Fundamentals**  
|            |                          | **Media**  
|            |                          | - Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores  
|            |                          | - Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation  
|            |                          | **Channels**  
|            |                          | - Explicit party-to-party communication; naming  
|            |                          | - APIs: sockets, architectural and language-based channel constructs  
|            |                          | **Memory**  
|            |                          | - Architectures in which parties directly communicate only with memory at given addresses  
|            |                          | - Consistency: Bitwise atomicity limits, coherence, local ordering  
|            |                          | - Memory hierarchies, locality: caches, latency, false-sharing  
|            |                          | - Heterogeneous Memory using multiple memory stores, with explicit data transfer across them; for example, GPU local and shared memory, DMA  
| Explain   | KA  | 2 |
- Multiple layers of sharing domains, scopes and caches
  - Data Stores
    - Cooperatively maintained structured data implementing maps, sets, and related ADTs
    - Varieties: Owned, shared, sharded, replicated, immutable, versioned

<table>
<thead>
<tr>
<th>PD</th>
<th>Communication</th>
<th>Programming with Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>● Using channel, socket, and/or remote procedure APIs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Using shared memory constructs in a given language</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PD</th>
<th>Communication</th>
<th>Properties and Extensions. <strong>One or more of:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>● Media</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Topologies: Unicast, Multicast, Mailboxes, Switches; Routing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Concurrency properties: Ordering, consistency, idempotency, overlapping with computation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Reliability: transmission errors and drops.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Data formats, marshaling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Protocol design: progress guarantees, deadlocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion), Responsiveness (liveness).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Applications of Queuing Theory to model and predict performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Channels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control</td>
</tr>
</tbody>
</table>

|    |              | Develop |
|    |              | CS |
|    |              | 1 |

|    |              | Explain |
|    |              | KA |
|    |              | 6 |
○ Program control for sending (usually procedural) vs receiving (usually reactive or RPC-based)
○ Formats, marshaling, validation, encryption, compression
○ Multiplexing and demultiplexing in contexts with many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs.
○ Formalization and analysis; for example using CSP

- Memory
  ○ Memory models: sequential and release/acquire consistency
  ○ Memory management; including reclamation of shared data; reference counts and alternatives
  ○ Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays
  ○ Emulating shared memory: distributed shared memory, RDMA

- Data Stores
  ○ Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains,
  ○ Faults and partial failures; voting; protocols such as Paxos and Raft
<table>
<thead>
<tr>
<th>PD</th>
<th>Coordination</th>
<th>Fundamentals</th>
<th>Explain</th>
<th>CS</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>Coordination</td>
<td>Programming with coordination</td>
<td>Develop</td>
<td>CS</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Controlling termination</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Security and trust: Byzantine failures, proof of work and alternatives</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Dependent actions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Execution control when one activity’s initiation or progress depends on actions of others</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Completion-based: Barriers, joins</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Data-enabled: Produce-Consumer designs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Condition-based: Polling, retrying, backoffs, helping, suspension, queueing, signaling, timeouts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Reactive: enabling and triggering continuations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Progress</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Dependency cycles and deadlock; monotonicity of conditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Atomicity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Atomic instructions, enforced local access orderings</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Locks and mutual exclusion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations, deadlock</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Using transactional APIs in a given framework

<table>
<thead>
<tr>
<th>PD</th>
<th>Coordination</th>
<th>Properties and extensions. <strong>One or more of:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- <strong>Progress</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Performance: contention, granularity, convoying, scaling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Non-blocking data structures and algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>Atomicity</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Ownership and resource control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Lock variants: sequence locks, read-write locks; reentrancy; tickets</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Transaction-based control: Optimistic and conservative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Distributed locking: reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Interaction with other forms of program control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Alternatives to barriers: Clocks; Counters, virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Speculation, selection, cancellation; observability and security consequences</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Resource-based: Semaphores and condition variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting</td>
</tr>
</tbody>
</table>

Explain KA 6
| PD | Software Engineering | Safety, liveness and performance requirements  
● Temporal logic constructs to express “always” and “eventually”  
● Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements | Explain | CS | 1 |
| PD | Software Engineering | Identifying, testing for, and repairing violations  
● Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock). | Evaluate | CS | 1 |
| PD | Software Engineering | Specification and Evaluation. **One or more of:**  
● Formal Specification  
  ○ Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications  
  ○ Use of tools such as UML, TLA, program logics  
  ○ Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting  
● Static Analysis  
  ○ For correctness, throughput, latency, resources, energy  
  ○ dag model analysis of algorithmic efficiency (work, span, critical paths)  
● Empirical Evaluation  
  ○ Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, continuous integration, | Evaluate | KA | 3 |
| PD | Algorithms and applications | Expressing and implementing parallel and distributed algorithms  
- Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs | Develop | CS | 1 |
| PD | Algorithms and applications | Survey of primary categories and algorithmic domains (listed below). | Explain | CS | 1 |
| PD | Algorithms and applications | Algorithmic Domains. **One or more of:**  
- Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning  
- Data processing: sorting, searching and retrieval, concurrent data structures  
- Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics  
- Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms  
- Logic: SAT, concurrent logic programming | Develop | KA | 9 |
- Graphics and computational geometry: Transforms, rendering, ray-tracing
- Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms, Batching, prioritization, partitioning, decentralization via work-stealing and related techniques
- Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.