Parallel and Distributed Computing

[Notes for viewers and commenters:

This is an early draft of a section of the upcoming ACM/IEEE/AAAI Computer Science curriculum recommendations, to eventually replace CS2013 at https://www.acm.org/education/curricula-recommendations. The “CS core” here and elsewhere deals with minimal coverage of concepts and skills for every CS graduate. The “KA core” extends these for any CS degree program with documented coverage of the Parallel and Distributed Computing Knowledge Area (whether required, elective, or some combination). You can find snapshots of early drafts of other KAs at https://csed.acm.org/.

You can help improve this document in any of three ways:

- Add comments to this document
- Send mail (dl@cs.oswego.edu) with comments and suggestions
- Send mail asking to be an anonymous reviewer, which I will forward to CS202X to solicit an anonymized review.

The theme is “parallel/distributed/concurrent computing from bits to bitcoin”, in the form of specifications, not implementations (courses). The main issues and questions are whether there are missing, unnecessary, or mis-stated concepts or skills in the CS and KA core.

]

Preamble

Parallel and distributed programming remove some simplifications and restrictions about programs made in sequential programming, revealing further distinctions, techniques, and analyses that appear at each level of computing systems. Parallel and distributed programs operate nearly everywhere, nearly all the time. The simplified views of programming in strictly sequential languages and platforms often “leak”, revealing at least some of the underlying structure and properties of systems and software that every CS graduate should understand, as indicated in PD CS Core requirements.

Increasing parallelism is an inevitable consequence of increasing numbers of gates in processors, processors in computers, and computers everywhere, all of which may be operating at the same time. Parallelism generally implies some form of distribution because multiple activities occurring at the same time must 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-based constraints. Parallel and distributed programming techniques are usually intended to improve performance compared to sequential programs, but also cope with the
intrinsic concurrency and interconnectedness of the world, including cases in which some components or connections fail or misbehave.

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. To further clarity coverage, this document also includes a section fleshing out terminology mentioned in the topic outlines.

Allocation of Core Hours

PD. Parallel and Distributed Computing (8 CS Core hours, 30 KA Core hours)

<table>
<thead>
<tr>
<th></th>
<th>CS Core hours</th>
<th>KA Core hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD/Fundamentals</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>PD/Program Design and Execution</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>PD/Communication</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>PD/Software Engineering</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>PD/Algorithms and Application Domains</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

CS Core coverage establishes familiarity with concepts, and the ability to construct and evaluate programs. CS Core Learning Outcomes include constructing a program (in a choice of several styles) with multiple activities and analyzing basic properties. These will often be covered as part of a core curriculum, extending coverage included in SDF, SF, PL, and AL, but also serve as preliminaries in PD-related courses.

KA Core coverage includes more depth, and more opportunities for experience-based learning. Students who complete the KA Core should understand, develop, and evaluate parallel and distributed software across a range of contexts, usually with more depth of understanding and experience in only some of these. This document differs from predecessors by refactoring KA content to equally address different approaches to parallel and distributed programming, while still accommodating the different ways in which courses and curricula may be primarily focused. The borders among these are not always clear-cut; all could be included in a single course, or offered across several different courses, each with more depth than specified. However, no topics are explicitly labeled as elective, but require only basic familiarity for those falling outside any primary focus. Also note that some topics (including those with explicitly indicated overlap) could be included in courses primarily covering other KAs such as operating systems and networking.

Description of Knowledge Units
PD/Fundamentals [8 CS Core hours]

- **Topics**
  - Program design and execution fundamentals
    - Parallel decomposition and composition
    - Procedural, reactive (event-driven), and distributed programming
    - Mappings to CPUs, SIMD devices, hosts; emulation
    - Nondeterminism and Scheduling
    - Survey of interactions with other forms of program control – sequencing, conditionals, iteration, call/return, and exceptions.
  - Communication Fundamentals
    - Channels, shared memory, IO, data stores
    - Available and/or preferable forms of communication rely on the nature of underlying hardware, connectivity, language support, and protocols
    - Consensus
    - Conflict: interference, contention, data races, ordering dependencies
    - Coordination: blocking and alternatives
  - Software Engineering Fundamentals
    - Specification, analysis, testing, and evaluation concepts and techniques extend sequential counterparts by addressing parallel actions across places
    - API and component design: Isolating vs explicitly parallel / distributed
    - Correctness: Safety and liveness specifications and analyses; common forms of programming errors including failure to ensure necessary ordering (race errors), atomicity (check-then-act errors), or termination (livelock).
    - Security: correctness in the presence of unprogrammed concurrent activities
    - Performance Goals and Metrics: Throughput, responsiveness, latency, availability, energy consumption (sustainability), scalability, resource usage, communication costs, waiting and rate control
  - Survey of programming, algorithms, and applications, including:

<table>
<thead>
<tr>
<th>Category</th>
<th>Typical Execution agents</th>
<th>Typical Communication protocols</th>
<th>Typical algorithmic domains</th>
<th>Typical primary performance goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore</td>
<td>threads</td>
<td>Shared memory, Atomic synchronization</td>
<td>Resource management, data processing</td>
<td>(vary)</td>
</tr>
<tr>
<td>Reactive</td>
<td>Handlers, threads</td>
<td>IO Channels</td>
<td>Services, real-time</td>
<td>latency</td>
</tr>
</tbody>
</table>
Data parallel | GPU, SIMD, accelerators, hybrid | Shared memory, messaging | Linear algebra, graphics, data analysis | throughput, energy
---|---|---|---|---
Cluster | Managed hosts | Sockets, Message channels | Simulation, data analysis | throughput
Cloud | Provisioned hosts | Service APIs | Web applications | scalability
Distributed | Autonomous hosts | Sockets, Data stores | Fault tolerant data stores and services | reliability

- **Illustrative Learning Outcomes**
  - Show how to parallelize a compound numerical expression
  - for example: $a = (b+c) \times (d + e)$.
  - Identify a data race in a given program
  - Show how to avoid or repair a race in a given program
  - In a given context, explain the extent to which introducing parallelism in an otherwise sequential program would be expected to improve throughput and/or latency, and how it may impact energy efficiency
  - Determine whether shared memory or message passing would be preferable for a given application in a given context
  - Write a data-parallel program that computes the average of an array of numbers.
  - Write a program that correctly terminates when all of a set of concurrent tasks have completed.
  - 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).
  - Write a producer-consumer program in which one activity generates numbers, and another computes their average

**PD/Program Design and Execution [9 KA core hours]**
- **Topics**
  - Forms of parallel actions: granularity, functions, closures, side-effects, sessions, services, transactions
  - Provisioning and mapping to computational units
  - Activation (deploying and initiating parallel actions)
  - Mappings and mechanisms: Internal CPU data parallelism, Heterogeneous data parallelism, Task parallelism, Actors, Multiplexed (scheduled) concurrency, Clusters, Distributed systems; emerging technologies including quantum computing and molecular computing
  - Scoping: Structured before/after constructs, sessions
○ Sequencing: Clocks, Counters and virtual clocks. Consensus, Dataflow and continuations. Join-based Futures and RPC, Gathering results (reducers, collectors).
○ Conditionals: Speculation and cancellation
○ Control flow: Scheduling computations, Series-parallel loops, Pipelines and Streams, nested parallelism.
○ Exceptions and failures. Handlers, detection, timeouts, fault tolerance.

● Illustrative Learning Outcomes
○ Write a filter/map/reduce program in multiple styles
○ Write a program in which the termination of one set of parallel actions is followed by another upon completion
○ Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
○ Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others
○ Write program for multiple hosts to agree upon the current time of day; discuss its limitations compared to protocols such as NTP
○ Write a service that creates a thread (or other procedural form of activation) to serve (for example a web page) to each new client.

PD/Communication [9 KA core hours]
● Topics
○ Connection properties
  ■ Topologies: Unicast, Multicast, Mailboxes, Switches
  ■ Concurrency properties: Ordering, consistency, idempotency, reliability
  ■ Protocol design: progress guarantees, deadlocks
  ■ Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion) Responsiveness (liveness).
○ Message Passing Channels
  ■ Explicit party-to-party communication; naming, routing, overlays
  ■ Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control
  ■ APIs: sockets, language-based channel constructs
  ■ Program control for sending (usually procedural) vs receiving (usually reactive or RPC-based)
  ■ Message Data Formats, marshaling, encryption, integrity checking, compression.
○ Shared Memory
  ■ Multicore architectures in which parties directly communicate only with memory; addressing, sharing domains; hybrids including DMA
  ■ Memory Consistency: Bitwise Atomicity limits, Coherence, Local ordering, language-level memory models.
- Memory locality: caches, false-sharing
- Preventing data races: Atomic updates, Enforced ordering, locks, transactions

- IO Device Communication
  - Multiplexing and demultiplexing communication with possibly many relatively slow IO devices
  - Completion-based and scheduler-based techniques; async-await, select and polling APIs.
  - Applications of Queuing Theory to predict and control performance

- Data Stores
  - Jointly maintained data structures, including key-value stores and other databases
  - Categories: Shared memory, owned, sharded, replicated, immutable, versioned
  - Faults and partial failures
  - Consistency and consensus: atomicity, coherence, causal ordering, conflict resolution, eventual consistency, blockchains, protocols such as Paxos and Raft
  - Security and trust: Byzantine failures, proof of work and alternatives

- Coordination
  - Options: Waiting, polling, retrying, helping, timeouts
  - State-based, data-based techniques
  - Progress properties including lock-free, wait-free, fairness

- Illustrative Learning Outcomes
  - Write a program that distributes different segments of a data set to multiple workers using message passing, and collects results (for the simplest example, summing segments of an array).
  - Write a parallel program that requests data from multiple sites, and summarizes them using some form of reduction
  - Compare the performance of buffered versus unbuffered versions of a producer-consumer program
  - Determine whether a given communication scheme provides sufficient security properties for a given usage
  - Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
  - Give an example of a scenario in which blocking message sends can deadlock.
  - Describe at least one design technique for avoiding liveness failures in programs using multiple locks
  - Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
  - Give an example of a scenario in which an attempted optimistic update may never complete.

PD/Software Engineering [3 KA Core hours]
- Topics
○ Specification: Extensions of sequential requirements such as linearizability; protocol and session specs, safety, liveness, security
  ■ Temporal logic constructs to express “always” and “eventually”
  ■ Use of tools such as UML, program logics, languages such as TLA
○ Static Analysis: Correctness, throughput, latency, resources, energy; dag model analysis: work/span, Amdahl's law;
○ Dynamic Analysis: testing and debugging; tools such as race detectors
○ Performance evaluation: Measuring throughput, overhead, waiting, contention, communication, data movement, locality, resource usage

● Illustrative Learning Outcomes
○ 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
○ Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
○ 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.

PD/Algorithms and Application Domains [9 KA Core hours]
● Note: Nearly every problem with a sequential solution also admits a parallel and/or distributed solution; 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 engineering too large to summarize here. Listed topics survey ideas, properties, and range of applicability. In-depth (typically project-based) coverage of a subset of these is expected to vary according to the focus of a course. Doing so will also typically include coverage of more specialized analysis and evaluation techniques
● Topics
  ○ 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, transactions, multiplayer games.
● Illustrative Learning Outcomes
○ 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.

○ Critique the design and implementation of an existing component or application, or one developed by classmates

○ Compare the performance and energy efficiency of multiple implementations of a similar design; for example multicore versus clustered versus GPU.

**Terminology and background**

**Program design and execution**

- **Parallel decomposition** is a declarative technique extending the notion of commutativity – that the order of executing two sub-operations doesn’t matter with respect to a given property – to further (permissively) assert that they are independent and may be performed (nondeterministically) in no particular order: at the same time, or in an order that may appear to be different across different observers. For parallel actions A and B, it is not the case that A must happen before B, or vice versa. A **race** (when restricted to accesses to a single variable, a “data race”) is a design or programming error where an ordering is required for correctness but not ensured by a program. A sequential requirement in an otherwise parallel program may be expressed as the **dependency** of one action on completion of one or more other actions. Some notations and languages by default assume lack of ordering unless dependencies are explicitly indicated. Most forms of parallelism that are implicit or declarative at one level of programming become explicit at lower levels.

- **Procedural** parallel programming constructs explicitly arrange new **parties** (parallel activities in the form of threads, processes, hosts, or other devices or circuitry) to become activated at specified program points. **Placement** and mappings of activities to underlying computing resources may be explicit, or delegated to other system software.

- **Reactive (event-driven)** parallel (concurrent) programming constructs arrange that parallel activities become initiated (or resumed) by the actions of the environment, including clients, users, sensors, IO devices, or other agents, by establishing handlers or related programming constructs to be invoked (aka triggered) upon the occurrence of an **event** – for example a processor interrupt, a message from an external device channel, client input on a socket, or a synthetic internal event. An **async** reactive handler in turn procedurally activates one or more parallel actions. However, when most activities are externally triggered, concurrent programming is often more focused on controlling interactions of the component actions of these activities rather than procedurally arranging more of them.

- **Distributed** programming constructs explicitly **place** or otherwise arrange or allow parallel activities to occur on hosts or devices that may be untrusted because they are not under the control of any one program or administrative domain, with the goal of maintaining services even when some participants fail or misbehave.
- **Consistency** constraints control whether one party or observer of a program execution might see one of two events or effects programmed in parallel occurring before the other, when another observer might see a different order.

- **Emulation.** A trace of a program execution forms a rooted directed acyclic graph (dag, pomset), not just a sequence. Any dag can be topologically sorted, resulting in one (of usually many possible) linearization sequences; for example by picking at each step one of the next possible events in some arbitrary order. So, at least in this sense, a parallel program can be emulated by a sequential program in which events may be apparently reordered with respect to some expectations. However, the properties of a strictly ordered emulated program are not necessarily identical to partially-ordered parallel executions in which two events need not have any (observable) order.

- **Scheduling.** Emulation may be extended to execute a program with a lot of parallelism with fewer parallel execution resources. Conversely, some sequential programming loops can be emulated using parallel (SIMD) replication, which is an increasingly common compiler optimization. (Note: the term “scheduling” may also refer to arranging the order of explicitly series/parallel computations.)

- **Parallelizing Actions.** Just about any computation can be programmed in parallel, with granularity ranging from a single bit operation up to an entire application program or service. Actions (even entire programs) may be characterized as closures/lambdas with arguments, contexts, and bodies. In most contexts, a parallel action constitutes a possibly-transient role of a parallel execution component (party). Categories of parallel actions extend those from common programming constructs. These include functions producing results available upon completion, procedures performing side effects that are usable upon completion, those primarily sending or receiving data, and/or those performing nested parallel operations. **Service** (aka Actor or daemon) activities include indefinitely looping bodies that may only terminate exceptionally; for example those that repeatedly input data and output transformed data. Some actions may be specified to be atomic (versus “composite”) with conceptually all-or-none instantaneous results or effects with respect to observers at a given programming level. In layered systems, a common goal is to make composite actions appear to be atomic at a higher layer. Atomicity extends the notion of data structure invariants. Example: in a simple array-based stack, the stack-pointer sp should always be equal to the number of elements. However, in a push operation array[sp++]=x, the invariant is momentarily broken (“inconsistent”). Techniques that prevent this from being observable are usually called “conservative”; those that allow observability but prevent unwanted consequences are “optimistic”. A joint action intrinsically involves multiple activities to achieve a common side effect. A common goal is to provide **transactionality** -- atomicity with respect to other observers; for example transferring money from one account to another.

- Like sequential decomposition, parallel decomposition may be recursive. For example, in array-based data-driven applications, by splitting data ranges with respect to array indices, ideally balancing parallelism across available processors, to minimize depth of critical paths while still avoiding overhead that would not make parallelism worthwhile.

- **Activation** of a parallel activity may be arranged **procedurally**, by **forking** (aka spawning) one at some point in a program. Or **reactively**, by establishing a **handler** to
be invoked upon the occurrence of an event. An async reactive handler in turn procedurally activates one or more parallel actions.

- **Provisioning** is the construction, allocation, or reservation of computational resources (which need not all be of the same kind) to execute a parallel/distributed program. Provisioning may be constrained by available hardware, or system policies, may be dynamic or elastic, and/or may entail emulation via scheduling or alternative mappings. A system or framework may further limit the total number of concurrently active actors components, sometimes to only one (as in many GUI frameworks) performed by an event loop thread, requiring or allowing others to be queued.

- **Activation.** Whether procedural or reactive, there are up to three steps for initiating an action with body B, that may be seen as variants or extensions of those required for a sequential function call. These steps may be extended to cases with higher fan-out -- introducing more than one parallel action at a time. The details vary widely across system layers and programming models, including the following (with many mixtures and in-between cases)
  - **Mapping:** Constructing or reusing a component to perform the action. This component may have a usable distinct identity, and membership in a distinct group.
  - **Configuration:** If necessary, supplying context and parameters, which may involve some form of communication (discussed below).
  - **Triggering:** Starting or enabling the activity, which may require scheduling it to run in the future if the component is not yet available.

- **Internal CPU and SIMD data parallelism.** In CPUs, a finite number of components are available to perform a fixed set of parallel operations at the machine word level. They take their arguments as data signals, and are enabled via gates. For the simplest example, a 64bit AND instruction enables 64 parallel AND gates each with their bit of data, rather than a sequential 64-iteration loop -- in data parallel hardware, loops are replaced by replication under parallel activation. This bitwise data parallelism is an example of “SIMD” (single instruction multiple data) parallelism. However, some such instructions need not be strictly SIMD. For example addition, where a carry in one adder affects results of another. CPU designs include ways to do this in fewer than O(wordSize) sequential steps. CPU internals also include other forms of parallelism.

- **Heterogeneous data parallelism.** The ideas of bitwise parallelism extend to vector parallelism (operations on adjacent data) common in heterogeneous extended instruction sets, GPUs, and FPGAs. Operations include forms of map, reduce, and select, as well as those supporting linear algebra functions such as matrix multiplication. Usages typically rely on a fixed set of processing units, but require data manipulation to configure each to operate on its associated data. “Data parallelism” also encompasses “bulk” parallelism -- operations on collections of elements not necessarily structured as arrays, that may be less amenable to vector-based techniques.

- **Task parallelism.** Each of the cores in a system plays the role of an executor: A virtual-CPU (basically, a universal Turing Machine) that repeatedly accepts a submitted task (often as a closure) and performs it. Submission operations may return references (or ids) for Future, Thread, or Process objects with APIs allowing inspection and control.
A processor (SMT or Hyperthreaded), operating system, hypervisor, virtual machine, and/or language run-time system then maps tasks to processing units, often at multiple levels; for example mapping tasks to user-level threads to system-level threads to cores.

- **Actors.** Reactive event driven components may be structured as objects (actors, aka daemons) maintaining state that persists across invocations. The primary programming steps require system and language support for triggering actions on each event occurrence, which may rely on state machine design to associate transitions with events.

- **Multiplexed (scheduled) concurrency.** Whether generated explicitly by forking actions, or implicitly by establishing reactive components, there may be many more runnable tasks than cores to run them. Aggregate throughput may improve by multiplexing rather than permanently binding tasks at each system level. When a faster activity communicates with slower ones (including unprogrammed external concurrent agents), awaits input, or contends for shared resources, it may spend most of its time idly waiting for events, reducing effective parallelism. In such cases, a processor or system software may context-switch among tasks when any of them block waiting for events, choosing which one to run next with respect to a scheduling policy, and re-enabling blocked activities when enabling events occur. Alternatively, a program may divide actions into subtasks that never (or rarely) block, but instead establish a reactive continuation task triggered when the enabling event occurs. Either of these approaches may add overhead but may improve aggregate performance. Further options include preemptive multiplexing in which a long-lived task may be run for a while, then artificially suspended (for example via clock interrupts), swapped out, and later resumed, which may improve latency for other tasks, at the cost of poorer scalability under increasing task load as overhead dominates.

- **Clusters.** Each node of a typically finite cluster of computers is maintained under a common operating system, framework, or language run-time system, allowing remote activation of tasks, usually with explicit assignment of different roles for different nodes, each with a given identity (address or index). Communication among processes on different nodes is restricted to mechanisms that do not directly require per-system shared memory.

- **Distributed computing.** Each node of a typically unbounded number of computers plays the role of a virtual computer system, normally identified by a network address. In cloud-based systems, actions are typically enabled using extended forms of closures that include configuration scripts.

- **Emerging technologies.** **Quantum** computers simultaneously maintain all possible states of a set of bits, so are intrinsically parallel. **Molecular** programmable Chemical Reaction Networks arrange parallel computations among billions of agents, usually with only indirect control over sequencing actions. Both are the target of research and experimentation. As of this writing, it is premature to include detailed curriculum recommendations.

- **Sequencing.** Completion of a (parallel) action in general causes something else to happen next; even if the “something else” is just program termination. The steps are complementary to those used to initiate activities, and may be intertwined with them.
when one parallel activity leads to another. These may be extended to cases with higher
fan-in – triggering only upon competition of multiple activities. Underlying mechanics
can take several forms. Techniques based on possibly-virtual clocks are usually termed
“synchronous”, the others “asynchronous”
  ○ Data collection. If necessary, recording any results.
  ○ Resource Management. If necessary, making the component available for future
    use
  ○ Triggering. Enabling actions dependent on completion

- Clocks. For most bitwise parallel instructions, CPUs use an indirect approach to
  sequencing. Require that each action completes within each clock tick, and then trigger
  the next one. Thus, each tick serves as a barrier, separating instructions sequentially.
  Without further mechanics, this approach restricts actions to those that can complete per
  tick.

- Counters and virtual clocks. A clock is a form of global (time-stamp) counter. Counters
  can be disentangled from real-time by advancing them at the ends of actions, and need
  not be globally scoped. An “acknowledgement” serves as the limiting case of a virtual
  clock with maximum count one. Virtual clocks can be used to advance programs based
  on counts serving as barriers across “phases”, “generations”, or “epochs” (which
  correspond to common APIs and/or language constructs), as well as those separating
  steps of “virtually synchronous” languages and frameworks. Agreement across parties
  on the current value of a shared counter is among the most basic consensus
  mechanisms. These are especially useful with multiple fan-ins, but become more
  challenging in distributed contexts accommodating faults (see below). Virtual clocks may
  be periodically synchronized, within some tolerance, with real-time clocks, using
  protocols such as ntp.

- Consensus. The idea of counters can be extended with further processing to trigger
  activities based on common agreement of just about any criterion. For example, program
  termination in task-based systems may rely on consensus that there are no more tasks
  to be performed. In systems primarily providing services based on a shared data store,
  consensus is most often applied to predicates about the datastore (for example
  transaction history), but further interacts with consistency requirements discussed below.
  More generally, this is the primary focus of concurrent constraint programming.

- Speculation, cancellation, and Short-circuiting. Multiple parallel functions may be
  established even when only one will be eventually used. Speculation techniques
  perform both of the two possible actions in an if/else conditional. Unused processing
  may then be canceled and/or ignored, This requires “any” or “OR” style versus “all” or
  “AND” style sequencing control, and may improve throughput or latency at the cost of
  additional energy.

- Dataflow. A continuation action may be associated with a parallel action, triggered
  upon completion. In CPUs, this is the main idea behind Instruction level parallelism (ILP).
  Because each continuation/completion action is triggered when data is available,
  dataflow computation is intrinsically dependency-based, not globally sequenced
  (although these are often intermixed). However, in strictly sequential programs, users
  cannot tell the difference, at least upon program termination. Explicit, higher-level forms
of dataflow include Promises and Completions, available in various languages, libraries, and frameworks.

- **Joins (Futures).** As control-flow constructs, parallel activations and completions present programmability issues similar to those of goto statements (which they may be considered an extended form of). To mesh better with sequential control constructs, completion of an action may be handled analogously to a function/procedure call, resuming a computation being accessed on by a caller; typically the initiator of the activity. This “pull” style can be implemented via dataflow techniques in conjunction with mechanics that suspend the caller until the action is completed, then resume it, relaying the function results. In non-local systems, this is known as remote procedure call (RPC), remote method invocation (RMI), and similar terms. Some languages and frameworks support only a resultless form of join, requiring that users deal with computation results themselves.

- **Gathering results.** Processing of the results of a set of parallel functions controlled by any of the above techniques include **collectors** that place them in a common data structure, and **reducers** that combine them into a lower-dimension value, for example a sum. Under tree-structured nested parallelism, reduction may be performed in parallel, “up” the computation tree. Collectors may require selection of only some results from each computation, possibly accomplished using compressed-store or parallel-prefix techniques, or using concurrent data stores.

- **Scheduling computations.** Parallelizing sequential code (versus designing code to be parallel in the first place) often requires transformations to establish independence and/or specially deal with non-independence (usually with a loss of some parallelism). Tool-assisted parallelism attempts to discover such cases and transformations to produce parallel code automatically.

- **Scoping.** Most languages and/or their APIs and frameworks provide **session** scoping constructs that confine a set of possibly-asynchronous parallel actions in a before/after fashion, sometimes with multiple pathways. Structured constructs and APIs such as async-finish, spawn-sync, forEachInParallel, and invokeAll support activation of a set of actions, usually with a specified after-action upon their completion, an opportunity to catch and process exceptions across them, and/or a means of cancelling unneeded remaining work. Component designs and APIs that restrict parallel/distributed activities within such scopes and sessions are usually simpler to use and compose.

- **Pipelines and Streams.** If a program component takes the form of a (typically unbounded) loop iterating over a series of transformative steps, each requiring significant independent computation, the processing of each step may be performed in parallel, with a lag of one iteration unit per step as the **producer-consumer** pipeline fills. **Reactive streams** take the same form, but express the flow in terms of completions or callbacks.

- **Nested parallelism.** A top-level parallel decomposition may include sub-parts that may be further parallelized; for example, the merge step of a parallel merge sort. These may require more steps to map to parallel hardware (which does not in general physically nest).
• **Series-Parallel Loops** arise when a set of parallel actions form the body of a sequential outer loop, for example in a simulation program where each iteration (sometimes called a phase, generation, or epoch) models a time step that is used as the basis of the next iteration. These **bulk-synchronous** designs may be programmed using extensions of count-based control including barrier-based constructs that provide not only per-iteration count-based control, but also a step or phase number, as well as a means for checking outer loop termination. Some algorithms are further amenable to streamlining by reducing the need for full-barriers on each iteration.

• **Exceptions and failures.** Exceptional completion (or non-completion) of an activity may trigger non-sequential processing. Cases include encountering exceptional events, incorrect computations, abrupt termination, and failure to detectably do anything at all. An exception occurring in one parallel action may require notification or cancellation of another. A common strategy to reduce the possibility of not handling such cases correctly is to employ finalization actions within code bodies or execution wrappers.

• **Failure Detection.** It might not be possible to control or distinguish failures in actions by parallel execution units versus failures in communication (**partitions**) or other infrastructure components. In such cases, a common ingredient of most distributed systems is to arrange heart-beat failure detection protocols to help determine reachability independently of task execution.

• **Timeouts.** When failures involve or prevent sequencing mechanisms, consensus-based continuation actions might not occur, leading to liveness failures. A first step in guaranteeing eventual responsiveness is to impose timeouts, which are only heuristically effective, since they cannot distinguish very slow processing, out-of-order communication, and permanent failure. When all three are possible, it is otherwise impossible to guarantee responsiveness or availability (FLP).

• **Fault tolerance.** A program or system may be required to continue functioning even when one or more parties maliciously or accidentally fail to conform to such specifications, by either ceasing operations (**fail-stop**) or performing arbitrary actions (**Byzantine failure**). However, there are intrinsic limitations in the ability of any system to do so. When there are at least a minimal number of correctly functioning and connected parties, protocols including Paxos and PBFT may be used to maintain service and availability requirements.

### Communication

• **Unicast** (one-to-one, aka point-to-point). A channel connecting two parties, In the simplest and most common form (half-duplex or producer-consumer channel) one party only sends data (messages), and the other only receives data (although possibly with synchronization communication by both).

• **Multicast** (one-to-many, aka multiplexers, publishers). Data messages issued by a producer are sent (usually identically) to multiple consumers, implicitly or explicitly forming a **group**. Multicast can be emulated by multiple unicasts (one sense of the term “overlay”), at the expense of added time to issue multiple messages, and can be decentralized by creating trees. (Similar emulations exist for other structural forms.)
- **Mailbox** (many-to-one, aka demultiplexers, collectors). Data Messages issued by multiple producers are all received (perhaps in no particular order) by a single consumer. There may be some means for receivers to identify the source or type of messages and handle them accordingly, and/or for messages to include some form of dispatch information. There may be some means to avoid denial of service attacks in which too many clients send requests.

- **Switches** (many-to-many, aka buses, crossbars). Any party may send a data message to any other party. Graphs of linked switches serve as routers.

- Concurrency properties of any communication mechanism include: Can messages or operations be issued observably out of order? Can different parties receive or process them in observably different orders, or not receive them at all? Are effects guaranteed to be **idempotent** – with a single effect no matter how many more times they are issued or received?

- **Secure** communication among entities in different places intrinsically involves physical processes that may be subject to errors, interception, and/or manipulation. In open systems, even some of the processes logically comprising a program or service may be untrusted. When some parties or channels are not under sole control of a program, systems must devise threat models that form the basis for such safety requirements, to ensure:
  - **Integrity.** Ensuring (with high probability) that the data received matches the data sent. Techniques include checksums and (secure) hashing.
  - **Privacy.** Ensuring (with high probability) that any third party intercepting data will not be able to decode it, using encryption techniques as well as disallowing eavesdropping from lower-layer components.
  - **Authentication.** Ensuring (with high probability) that the endpoints connect to the intended parties, using trust mechanics such as public key repositories.
  - **Authorization.** Ensuring that only parties that are independently permitted to receive, send, or use data do so, using permission records or protocols.

- **Communication Performance** may be characterized by:
  - **Latency.** The time taken for a signal to reach the recipient; a function of the physical distance between parties, and the speed of signals through the connecting medium (electrons, light).
  - **Bandwidth** (throughput). A function of the density of bits through the medium (perhaps via intermediaries), as well as data translation (possibly including crypto) rates.
  - **Contention** (congestion). The need for multiple producers to wait for each other to transmit to a common consumer (perhaps indirectly so, for a switch); also a function of fairness under contention.
  - **Responsiveness.** (liveness). Protocols may include synchronization, integrity checks or other obligations requiring that one party wait for acknowledgments or other responses from others. Lack of response from a party may lead to indefinite delays in others, or multi-party deadlocks.
- **Messaging Policies and protocols.** Language and API support for communication can take many forms. For remote communication, most systems support multiple forms of sockets. Other support varies in policy and usage across:
  - **Endpoints.** Parties may be denoted by network/port addresses, device IDs, memory addresses, processes, threads, indices, mapped service names, or other forms of identities. In secure protocols, some form of independent authentication is required to map parties to known entities.
  - **Sessions.** Some connections either physically or virtually “always” exist, while others must be explicitly opened before use and closed after use.
  - **Buffering.** Buffering reduces the impact of burstiness (jitter) when parties and/or the communication medium operate at different rates. Buffers are conceptually forms of bounded queues, often structured as circular array-based chunks of contiguous memory. Some systems support only unbuffered channels, or only buffered ones.
  - **Saturation** (waiting vs dropping). When a message send is attempted but the channel is not immediately available (and is unbuffered or saturated), some forms of channels (most notably UDP) will drop messages, while others cause senders to wait (perhaps until a timeout).
  - **Rate control.** Purely synchronous stop-and-wait protocols require that a sender block until an acknowledgement is received. Purely async protocols never block senders, even if messages are delayed or dropped. Windowing protocols such as TCP loosen the coupling of sending and acknowledgements, allowing higher throughput without drops.

- **Message Data.** The nature and format of transmitted data may vary across platforms, usages, and APIs. Hardware buses use native formats. Basic marshalling standards include conventions for transmitting records and variable-length arrays; these may vary according to encryption, integrity checking, and compression requirements. When endpoints do not share address spaces, pointers and references must be handled by remapping data structures with respect to different address base origins, typically as part of pickling or serialization utilities. More flexible, platform independent, but verbose schemes include JSON and XML. The efficiency of representations for bulk data often rely on how and when segments are communicated, for example Arrays-of-Structs vs Structs-of-Arrays. Choices among communication media, policies, and language/API support branch into a large set of possible programming approaches, and techniques. In practice though, there are a few common ones, including MPI, CUDA, RPC-based systems, and channel-based languages. These exist in addition to shared memory systems that differ in so many ways that they lead to distinct sets of programming techniques. However mixtures and extensions involving both forms are also common in the design and use of possibly-distributed or partitioned concurrent data stores.

- **Shared memory protocols.** In shared memory multicore, activities running on different cores (usually as threads mapped to cores) do not normally communicate with each other directly, but instead communicate with a memory system using switch/bus-based protocols mainly serving as memory-reads and memory-writes. In some systems and languages, only certain indicated programmer-visible locations/variables are available for
sharing across cores, but in most multicore-based platforms, any programmable, addressable location is potentially shared. In NUMA memory systems, a combination of processor-to-processor and processor-to-memory protocols may be required to access memory managed by one or more processors. Heterogeneous systems such as GPUs usually use a combination of explicit data communication and shared memory (sometimes via DMA).

- **Memory Consistency Properties.** Processor-to-memory communication is usually simpler and faster than processor-to-processor communication. However, main memory operates more slowly than CPUs. Memory protocols entail caching and buffering that is in part local to each core. At any given moment, two cores may have inconsistent views of the value of a given memory cell. Processors and their memory systems provide minimal guarantees that limit the kinds of inconsistencies that may be observed by programs (which are typically stronger than those guaranteed by non-point-to-point message passing protocols). These form the basis of a processor **memory model.** They almost always include the following:
  - **Bitwise Atomicity limits.** All of the bits of an access to a word-sized location are operated on as a whole. As of this writing, word-sized units are typically 64 bits, although there may be special memory instructions operating on shorter or longer sizes. Special rules applying to mixtures including sub-word-sized accesses such as writing only the low byte of a word, are less uniform across processors.
  - **Coherence.** For each main memory cell, accesses proceed in a linear order. This order need not be controllable by any given core, does not constrain the relative ordering of accesses across multiple variables, and may interact with caching and buffering that further weaken impact. However, under coherence, no matter how many threads or cores there are in a program, if they all try to write to the same location, the program runs at most as fast as the memory cell can be sequentially updated; this is one example of a sequential bottleneck in which multiple activities may all need to wait for another to complete. (Note: the lack of coherence guarantees or equivalents in most non-shared-memory systems is among the primary sources of programming differences among them.)
  - **Local ordering.** Every core sees its own reads and writes in a (partial) order consistent with the order that it issues them. Non-shared memory acts sequentially with respect to local sequential observers even though compiler optimizations and instruction-level parallelism can relax this ordering, under the constraint that these are not observable in strictly sequential programs.

- **Language-level consistency.** A language-level memory model provides rules for language constructs, mapped (possibly in different ways) to different processors. Among the strongest forms is **sequential consistency,** where allowed (strict) orders are those which a sequential emulation could produce. This is incompatible with the design of processors that rely on out-of-order processing, buffering, and speculation, and so is typically available only for specially marked variables. Weaker but widely useful alternatives supported by most languages include release-acquire consistency, and per-variable coherence.
● **Data Races.** Memory consistency rules are not nearly enough to guarantee sensible behavior in the presence of races in which multiple activities access shared variables, encountering Write-Write and Read-Write conflicts. For example if two threads T and U are both trying to perform “++count” for a shared variable count initially 0, both T and U could read 0 and write 1, thus “losing” an update.

● **Memory Locality.** Cores access memory using (usually multilevel) caches. Each cache is a set of cache lines holding a fixed number (for example 64 bytes) of contiguous data, read from and written to main memory, and evicted under some policy when new lines are needed. Execution is fastest when data being operated on is strictly local to a core’s local cache, and never accessed by others. It is nearly as fast when accessed read-only by all others. Other cases require cache consistency protocols that manage shared and exclusive access, with slower performance, especially in the case of false-sharing. In which a variable updated by one core happens to share a cache line with those heavily accessed by another core, Dealing with these kinds of problems may require low-level control over data layout even in high-level languages.

● **Atomic updates.** Multicore processors provide instructions (perhaps only applying to special kinds of variables) that atomically (indivisibly) read and (possibly) write. A common choice that suffices to address every coordination problem (a “universal consensus primitive”) is CompareAndSet (CAS; with variants sometimes known as compareAndSwap or compareAndExchange). In pseudocode:

```java
boolean compareAndSet(Location v, Value expectedValue, Value newValue) {
    if (value read at v equals expectedValue) { set v to newValue; return true; }
    else return false;
}
```

This operation must be atomic at the processor level. Otherwise it would be an example of a check-then-act error, where a condition is checked but does not necessarily hold during an action requiring it to still hold. Applied to the count problem, both threads could replace “++count” with:

```java
Int c;  do {  c = count; } while (!CompareAndSet(&count, c, c+1));
```

Most processors support additional atomic “RMW” (Read-Modify-Write) instructions as well, to simplify and speed up common usages. For example getAndAdd (fetchAndAdd) could be used here to avoid the loop.

● **Enforced Ordering.** Coherence alone makes no promises that accesses to two or more different shared variables take place in any relative order. Sometimes, such ordering is necessary for correctness. For example, if one thread creates a new object, then initializes it, and then assigns its reference to a shared variable p, the reference assignment must not be accessible before initialization completes. And similarly, any observer must see the current value of p (rather than some previous value) before dereferencing its fields. Processors either make special guarantees (as in x86) or provide special instructions (as in ARM) to support the need for this “safe publication” and related Release-Acquire processing such as RCU. Even stronger forms (C++ mode seq_cst, Java mode volatile) ensure two-way ordering needed for example when implementing blocking operations.

● **Exclusive access using locks.** Atomic ordering techniques primarily apply to communication and usage of single word-size scalars, or single pointers/references to
compound immutable data. But when programs deal with mutable structured data (objects, records) with multiple fields with possibly interdependent values that must obey invariants, they usually extend atomicity using protocols based on locking (for example a mutex). Each access (set of reads and writes) must be encased in lock/unlock before/after construction. (Failure to conform to this protocol is another form of data race.) In most systems, more than one form of lock is available, possibly including reentrant, read/write, and sequence locks. The longer a mutually exclusive lock is held, the more sequential bottlenecks occur for activities needing the lock. The use of locks introduces the possibility of deadlock, in which two or more parties each require two or more locks: without further coordination, it is possible for party P to hold lock A while trying to acquire lock B that is held by party Q while it is trying to acquire lock A, leading to total liveness failure. Without care, similar failures may be possible with other communication protocols.

- **Transactions.** Most forms of locks are designed to (conceptually) cover the fields of a single data object, record, table, or array. Transactions extend this to deal with atomic transitions across multiple objects. They may be either optimistic or conservative, or some combination. Hardware transactional memory (HTM) is generally optimistic: reading a set of values, performing computations on them, and committing results only there was no interference.

- **Data Stores** entail memory-based and communication-based schemes that ensure that some or all parties at least sometimes agree about values held in data stores representing discrete structures (sets, sequences, arrays, maps, matrices, graphs). In parallel and distributed systems, views of data stores may be construed as caches. Data stores may be immutable, in which case “updates” produce new data stores with new identities, or mutable, in which case each update may be (perhaps only conceptually) associated with a version tag, or log-structured, in which case an agreed-upon sequence of updates is maintained, perhaps in addition to techniques that allow determining current values without replaying the entire log. There are several basic strategies that may be based on any applicable communication technique; across each of them, users of a data store may maintain caches holding the last known data values.
  - Unowned non-distributed data stores rely on shared memory, perhaps emulated as distributed shared memory.
  - Owned data stores rely on a single owner party to provide an API and/or protocol for accessing and requesting to update data stores. It may be possible to transfer ownership.
  - Partitioned (sharded) data stores maintain different parts of a data store (for example, bins of a hash table, subtrees of a tree, or segments of an array) by different parties, that may all be known members of a designated group, or may be fully decentralized.
  - Replicated data stores redundantly maintain data stores across different parties that are all members of a designated group of replicates, normally in conjunction with a fault tolerance scheme. A replicated data store may have a designated leader.
Replicated log-structured (ledger-based) data stores maintain the entire history of transactions, along with validation techniques to ensure agreement about the sequence of transactions, and indexing to locate values or transactions.

- **Data store consistency** properties are based on algorithms and protocols providing some form of fault-tolerant consensus. Example protocols include Paxos and Raft, that can serve as the basis for distributed transactions managed by a designated group of hosts, usually serving as a replicated state machine. Many systems can tolerate forms of inconsistency if they are guaranteed to be transient. **Eventually consistent** data types (including CRDTs) employ protocols and frameworks that define merge operations for two or more versions of a data store that eventually produce consistent states. CRDTs can only be defined for data stores in which every possible update operation is mergeable. For example, add-only counters in which merges choose the maximum value across versions, and Sets with an “add-wins” rule in which merging a version with an element added with one in which it is removed, keeps the added element. Stronger forms include causal consistency, in which updates are only performed when parties are up-to-date with respect to the version the updates apply to. However, these are still weaker than full transactionality because they enable unresolved concurrent updates.

- **Trust.** Data stores maintained by mutually untrusted parties may require ledger-based designs in which any observer may validate any transaction in an agreed upon linear sequence or chain (or decide not to rely on the data store if not validated). Usually, multiple transactions are grouped together as blocks on each blockchain update. Validation may take the form of verifying a secure hash summarizing the content of each block and its order. In permission-based blockchains, only a fixed set of parties may perform updates, normally using Byzantine consensus protocols which succeed only when fewer than ⅓ of the participants fail, become unreachable, misbehave, or disagree. In permissionless systems, the contents of transactions are perhaps indirectly based on “coin” values; any party (“miner”) may attempt to update the chain, but must on average expend significant computation (proof of work) to do so, under an incentive system that produces coin value upon success. These forms of blockchains, and many variations, are the subjects of much recent commercial development and academic research. As of this writing, it is premature to include detailed curriculum recommendations.

- **Coordination.** An activity may not be able to proceed with an action due to the states of certain objects, global predicates, or other activities; for example situations analogous to stopping at a traffic light. The main options are:
  - **Blocking.** Suspending until conditions change,
  - **Polling.** Possibly doing something else and then rechecking
  - **Retrying.** Optimistically attempting to proceed anyway, subject to a subsequent check controlling whether to commit, abort, or repeat
  - **Helping.** Performing some action that will then allow progress by one or more threads.
  - **Timeouts.** Imposing time bounds on any of the above

- **Progress.** The extent to which coordination impacts global program liveness and progress properties can be categorized in terms of lock-based, obstruction-free,
lock-free, and wait-free properties, along with interactions with fairness – ensuring eventual progress of enabled actions.

- **Coordination techniques** include:
  - **State-based** coordination includes lock-based constructions such as monitors, and condition variables, semaphores, and barriers, as well as OS-level support such as park/unpark and futex constructions. Blocking options require that other activities notify schedulers or synchronization primitives in order to wake up blocked threads or activate continuation actions, which may require explicit arrangements by programmers.
  - **Data-based** coordination includes the use of data structures that may block, retry, or help in the course of data exchanges, such as blocking and non-blocking buffers, queues, or stacks. Some implementations extend state-based components to additionally include passing data; for example semaphore-based blocking queues.

**Software Engineering**

- Program- and component-specific parallel and distributed program specification and analysis techniques extend those for sequential programs. They must accommodate the lack of ordering guarantees during parallel execution and the lack of consistency guarantees among multiple observers. However, like sequential specifications, requirements almost always (perhaps implicitly) reference one or more initial and subsequent sequence points (for example to specify pre and post conditions). In the simplest cases, few extensions are required, for example when applied to known-to-be commutative actions on unobservable variables occurring between two given observable sequence points. Beyond this, most techniques are targeted to different execution and communication contexts for example, procedural vs triggered activation, protocol specs, memory consistency rules.

- Across these, several patterns of common errors are known, including check-then-act errors in which some condition is checked before engaging in some action without ensuring that the condition still holds during the action; for example checking whether an element of a collection exists before trying to add it, without ensuring that others are not also adding it.

- **Safety** properties assert that unwanted actions or effects never happen; in temporal logic: Always(not(predicate(...))). These extend notions of type safety and security to parallel contexts. In most programs, the main focus is on avoiding interference, where the actions of one activity may (perhaps only rarely) cause another not to attain intended outcomes, for example data-races in which the order of accesses to shared variables cannot be ensured. When parallelism is introduced across actions that were not initially designed to commute; programs must ensure that outcomes meet such thread-safety requirements when triggered independently and performed without ordering guarantees, or to ensure that these safety violations cannot occur.

- **Liveness** properties assert that required actions or effects eventually happen; in temporal logic: Eventually(predicate(...)). These extend the termination criteria of sequential programs to require that some or all activities complete. The most prominent
liveness failures are **deadlocks**, in which two or more activities endlessly wait for each other to proceed. Also **fault-tolerance** failures in which too many (perhaps all) parties and/or communications among them have failed. Many liveness requirements impose time limits that strengthen “eventually” properties. Safety alone, or liveness alone, are not typically useful for program specification -- a safety property tells you what must never happen, normally requiring liveness-based specifications to say what should eventually happen instead.

- **Throughput.** A set of operations may be executed more quickly if at least some of them are executed at the same time. Performance improvements typically reduce **sequential bottlenecks** by reducing the length of the **critical path** of sequential steps required for a computation. Throughput may be analyzed using dag model analyses, Amdahl's law, and related metrics.

- **Latency (Responsiveness).** Communication among entities (processors, memory, systems) occurring in different places intrinsically entails latency reflecting physical limitations of signals traveling through media across distances, which can sometimes be reduced by explicitly arranging greater **locality**. When a service or component has multiple concurrent clients or users, if each executes in parallel, then, in the absence of other constraints, clients need not **wait** for each other, reducing latency, and potentially improving **availability**, by ensuring that a failure in one service instantiation does not affect others. When a faster activity communicates with slower ones (including unprogrammed external concurrent agents), awaits inputs, or contends for shared resources, it may spend most of its time idly waiting for events, reducing effective parallelism. This may sometimes be addressed by multiplexing and scheduling. Latency goals and metrics may be extended to apply to not only average delays, but also their variance (**jitter**). In **real-time** programs and systems, most latency properties are included as requirements rather than performance goals, because acting too late is a form of failure.

- **Energy consumption.** Using more processing units and related resources at the same time requires more power and greater energy costs. In some cases, energy usage per unit time for a given computation or service stays approximately constant, or even improves, but some techniques that improve peak performance (for example speculation) may also decrease work-efficiency and overall energy efficiency. On the other hand, in some contexts, the main goal is maximizing **utilization** of available resources regardless of energy costs.

- **Scalability.** Ideally, throughput and/or latency metrics increase proportionally with the number of processing units. But even when programs are constructed to permit at least some parallelism, impact may vary due to **resource management**. Languages and platforms vary in the extent to which available computers, processing units, communication channels, and memory may be serially reused to support parallel programs, versus statically constructed or allocated. In addition to encountering overhead, when two or more activities both access another service, resource, or memory cell at the same time (for example, both trying to update a bank balance), the resulting **contention** control may force them to wait for each other. Languages and systems also
vary in the extent to which programmers (versus run-time systems) must perform static and/or dynamic resource allocation and reclamation.

- **Performance trade offs.** Performance properties may sometimes be traded off with respect to each other via techniques such as speculation, cancellation, locality, caching, bulk and buffered operations, non-work-efficient designs such as parallel-prefix, and reduced precision.

**List of Professional Dispositions Appropriate for this KA**

- TBD

**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
  - Optional, depending on application domains: 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, for example facility with Linear Algebra is needed to explain many high-performance algorithms

**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, See the partially out of date PD specific version: PD-relations with-other-KAs

**Subcommittee**

Chair: Doug Lea (State University of New York at Oswego)

Subcommittee members

- Sherif Aly (American Univ Cairo, lead for Networking KA)
- Michael Oudshoorn (High Point Univ, lead for Programming Languages KA)
- Qiao Xiang (Xiamen Univ, lead for System Fundamentals KA)
- Dan Grossman (Univ Washington, cs2013 PD lead)
- Sebastian Burckhardt (Microsoft Research)
- Vivek Sarkar (Georgia Tech)
- Maurice Herlihy (Brown Univ)
- Sheikh Ghafoor (Tennessee Tech)
- Chip Weems (UMass, also with NSF/IEEE-TCPP)

Other contributors: