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..

PDC has evolved from a diverse set of advanced topics into a central body of knowledge and practice, permeating almost every other aspect of computing. Growth of the field has occurred irregularly across different subfields of computing, sometimes with different goals, terminology, and practices, masking the considerable overlap of basic ideas and skills that are the main focus of this KA. 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.

Overview

The PDC KA is divided into five KUs, each with CS-Core and KA-core components that extend but do not overlap CS Core coverage that appears in other KAs.. They cover: The nature of parallel and distributed **Programs** and their execution; **Communication** (via channels, memory, or shared data stores), **Coordination** among parallel activities to achieve common outcomes; **Evaluation** with respect to specifications, and **Algorithms** across multiple application domains..

CS Core topics span approaches to parallel and distributed computing, but restrict coverage to those applying to nearly all of them. 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 needed as a basis for implementing and evaluating concepts and skills. They also avoid reliance on specifics that may vary widely (for example GPU programming vs cloud container deployment scripts), Prerequisites for PDC CS Core coverage include::

- SDF-Fundamentals. programs vs executions, specifications vs implementations, variables, arrays, sequential control flow, procedural abstraction and invocation, IO.
- SF-Overview: Layered systems, State machines, Reliability
- AR-Assembly, AR-Memory: Von Neumann architecture, Memory hierarchy
- MSF-Discrete: Logic, discrete structures including directed graphs.

Additionally, FPL (Foundations of Programming Languages) may be treated as a prerequisite, depending on other curriculum choices. The PDC CS Core requires familiarity with languages and platforms that enable expression of basic parallel and distributed programs. These need not be included in SDF or SF but are required in FPL-PDC, which additionally covers their use in PDC constructions. Also, PDC includes definitions of Safety, Liveness, and related concepts that are covered with respect to language properties and semantics in FPL. Similarly, the PDC CS Core includes concepts that are further developed in the context of network protocols in NC (Networking and Communication), OS (Operating Systems) and SEC (Security), that could be covered in any order..

KA Core topics in each unit are of the form “One or more of the following” for *a la carte* topics extending associated core topics. Any selection of KA-core topics meeting the KA Core hour requirement constitutes fulfillment of the KA Core. These permit variation in coverage depending on the focus of any given course. See below for examples. 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 information systems, but much less so in most data-parallel applications.

Changes since CS 2013

The PDC KA has been refactored to focus on commonalities across different forms of parallel and distributed computing, also enabling more flexibility in KA-core coverage, with more guidance on coverage options.

Core Hours

Knowledge Units	CS Core hours	KA Core hours

Programs	2	2
Communication	2	6
Coordination	2	6
Evaluation	1	3
Algorithms	2	9
Total	**9**	**26**

Knowledge Units

PDC-Programs

CS Core:

1. Fundamental concepts
    a. Ordering
        i. Declarative parallelism: Determining which actions may be performed in parallel, at the level of instructions, functions, closures, composite actions, sessions, tasks, services
        ii. Defining order: happens-before relations, series/parallel directed acyclic graphs representing programs
        iii. Independence: determining when ordering doesn't matter, in terms of commutativity, dependencies, preconditions
        iv. Ensuring ordering among otherwise parallel actions when necessary, for example locking, safe publication; and orderings imposed by communication: sending a message happens before receiving it
        v. Nondeterministic execution of unordered actions
    b. Places
        i. Devices executing actions include hardware components, remote hosts (See also AR-IO)
        ii. One device may time-slice or otherwise emulate multiple parallel actions by fewer processors (See also OS-Scheduling)
        iii. May include external, uncontrolled devices, hosts, and human users
    c. Deployment
        i. Arranging actions be performed (eventually) at places, with options ranging from from hardwiring to configuration scripts, or reliance on automated provisioning and management by platforms
        ii. Establishing communication and resource management (See also SF-Resources)
        iii. Naming or identifying actions as parties (for example thread IDs)
    d. Consistency
        i. Agreement among parties about values and predicates

ii. Races, atomicity, consensus
iii. Tradeoffs of consistency vs progress in decentralized systems

e. Faults
i. Handling failures in parties or communication, Including (Byzantine) misbehavior due to untrusted parties and protocols
ii. Degree of fault tolerance and reliability may be a design choice

2. Programming new activities
a. The first step of PDC-Coordination techniques. expressed differently across languages, platforms, contexts
b. Procedural: Enabling multiple actions to start at a given program point; for example, starting new threads, possibly scoping or otherwise organizing them in possibly-hierarchical groups
c. Reactive: Enabling upon an event by installing an event handler, with less control of when actions begin or end
d. Dependent: Enabling upon completion of others; for example, sequencing sets of parallel actions

KA Core:

3. Mappings and mechanisms across layered systems. **One or more of:**
a. CPU data- and instruction-level- parallelism (See also AR-Organization)
b. SIMD and heterogeneous data parallelism (See also AR-Heterogeneity)
c. Multicore scheduled concurrency, tasks, actors (See also OS-Scheduling)
d. Clusters, clouds; elastic provisioning (See also SPD-Common)
e. Networked distributed systems (See also NC-Networked-Applications)
f. Emerging technologies such as quantum computing and molecular computing

Illustrative Learning Outcomes

CS Core:

1. Graphically show (as a dag) how to parallelize a compound numerical expression; for example a = (b+c) * (d + e).
2. Explain why the concepts of consistency and fault tolerance do not arise in purely sequential programs

KA Core:

3. Write a function that efficiently counts events such as networking packet receptions
4. Write a filter/map/reduce program in multiple styles
5. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PDC-Communication

CS Core:

1. Media
a. Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores

    b. Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation (See also AR)
2. Channels
    a. Explicit party-to-party communication; naming channels
    b. APIs: sockets (See also NC-Introduction), architectural and language-based constructs
    c. IO channel APIs
3. Memory
    a. Parties directly communicate only with memory at given addresses, with extensions to heterogeneous memory supporting multiple memory stores with explicit data transfer across them; for example, GPU local and shared memory, DMA
    b. Consistency: Bitwise atomicity limits, coherence, local ordering
    c. Memory hierarchies: Multiple layers of sharing domains, scopes and caches; locality: latency, false-sharing
4. Data Stores
    a. Cooperatively maintained structured data implementing maps and related ADTs
    b. Varieties: Owned, shared, sharded, replicated, immutable, versioned
5. Programming with communication
    a. Using channel, socket, and/or remote procedure call APIs
    b. Using shared memory constructs in a given language

KA Core:

6. Properties and Extensions. **One or more of**:
    a. Media
        i. Topologies: Unicast, Multicast, Mailboxes, Switches; Routing via hardware and software interconnection networks
        ii. Concurrency properties: Ordering, consistency, idempotency, overlapping communication with computation
        iii. Performance properties: Latency, bandwidth (throughput) contention (congestion), responsiveness (liveness), reliability (error and drop rates), protocol-based progress (acks, timeouts, mediation)
        iv. Security properties: integrity, privacy, authentication, authorization (See also SEC)
        v. Data formats
        vi. Applications of Queuing Theory to model and predict performance
    b. Channels
        i. Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control
        ii. Program control for sending (usually procedural) vs receiving.(usually reactive or RPC-based)
        iii. Formats, marshaling, validation, encryption, compressIon
        iv. Multiplexing and demultiplexing many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs.
        v. Formalization and analysis; for example using CSP

c. Memory
    i. Memory models: sequential and release/acquire consistency
    ii. Memory management; including reclamation of shared data; reference counts and alternatives
    iii. 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
    iv. Emulating shared memory: distributed shared memory, RDMA
d. Data Stores
    i. Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains,
    ii. Faults, partitioning, and partial failures; voting; protocols such as Paxos and Raft
    iii. Design tradeoffs among consistency, availability, partition (fault) tolerance; impossibility of meeting all at once
    iv. Security and trust: Byzantine failures, proof of work and alternatives

Illustrative Learning Outcomes

CS Core:

1. Explain the similarities and differences among: (1) Party A sends a message on channel X with contents 1 received by party B (2) A sets shared variable X to 1, read by B (3) A sets "X=1' in a distributed shared map accessible by B.
2. 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:

3. 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).
4. Write a parallel program that requests data from multiple sites, and summarizes them using some form of reduction
5. Compare the performance of buffered versus unbuffered versions of a producer-consumer program
6. Determine whether a given communication scheme provides sufficient security properties for a given usage
7. Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
8. Give an example of a scenario in which blocking message sends can deadlock.
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks
10. Write a program that illustrates memory-access or message reordering.
11. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
12. Give an example of a scenario in which an attempted optimistic update may never complete.
13. Modify a concurrent system to use a more scalable, reliable or available data store

14. Using an existing platform supporting replicated data stores, write a program that maintains a key-value mapping even when one or more hosts fail.

PDC-Coordination

CS Core:

1. Dependencies
    a. Initiation or progress of one activity may be dependent on other activities, so as to avoid race conditions, ensure termination, or meet other requirements
    b. Ensuring progress by avoiding dependency cycles, using monotonic conditions, removing inessential dependencies
2. Control constructs
    a. Completion-based: Barriers, joins
    b. Data-enabled: Queues, Produce-Consumer designs
    c. Condition-based: Polling, retrying, backoffs, helping, suspension, signaling, timeouts
    d. Reactive: enabling and triggering continuations
3. Atomicity
    a. Atomic instructions, enforced local access orderings
    b. Locks and mutual exclusion; lock granularity
    c. Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers
    d. Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations
    e. Avoiding locks: replication, read-only, ownership, and nonblocking constructions
4. Programming with coordination
    a. Controlling termination
    b. Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races

KA Core:

5. Properties and extensions. **One or more of:**
    a. Progress
        i. Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability
        ii. Performance: contention, granularity, convoying, scaling
        iii. Non-blocking data structures and algorithms
    b. Atomicity
        i. Ownership and resource control
        ii. Lock variants and alternatives: sequence locks, read-write locks; RCU, reentrancy; tickets; controlling spinning versus blocking
        iii. Transaction-based control: Optimistic and conservative
        iv. Distributed locking: reliability
    c. Interaction with other forms of program control
        i. Alternatives to barriers: Clocks; Counters, Virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors

ii. Speculation, selection, cancellation; observability and security consequences
iii. Resource-based: Semaphores and condition variables
iv. Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.
v. Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting

Illustrative Learning Outcomes

CS Core:

1. Show how to avoid or repair a race error in a given program
2. Write a program that correctly terminates when all of a set of concurrent tasks have completed.

KA Core:

3. Write a function that efficiently counts events such as sensor inputs or networking packet receptions
4. Write a filter/map/reduce program in multiple styles
5. Write a program in which the termination of one set of parallel actions is followed by another
6. Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
7. Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others
8. Write a program for multiple parties to agree upon the current time of day; discuss its limitations compared to protocols such as NTP
9. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client

PDC-Evaluation:

CS Core:

1. Safety and liveness requirements (See also FPL-PDC:1)
    a. Temporal logic constructs to express “always” and “eventually”
2. Identifying, testing for, and repairing violations
    a. Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock)..
3. Performance requirements
    a. Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements (See also SF-Performance)
4. Performance impacts of design and implementation choices
    a. Granularity, overhead, energy consumption, and scalability limitations (See also SEP: Sustainability)
    b. Estimating scalability limitations, for example using Amdahl and Gustafson laws (See also SF-Evaluation)

KA Core:

5. Methods and tools. **One or more of:**

a. Formal Specification
    i. Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications
    ii. Use of tools such as UML, TLA, program logics
    iii. 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
b. Static Analysis
    i. Applied to correctness, throughput, latency, resources, energy
    ii. dag model analysis of algorithmic efficiency (work, span, critical paths)
c. Empirical Evaluation
    i. Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, visualizations, continuous integration, continuous deployment, and test generators,
    ii. Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads.
d. Application domain specific analyses and evaluation techniques

Illustrative Learning Outcomes

CS Core:

1. Revise a specification to enable parallelism and distribution without violating other essential properties or features
2. Explain how concurrent notions of safety and liveness extend their sequential counterparts
3. Specify a set of invariants that must hold at each bulk-parallel step of a computation
4. 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.
5. 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
6. Show how scaling and efficiency change for sample problems without and with the assumption of problem size changing with the number of processors; further explain whether and how scalability would change under relaxations of sequential dependencies.

KA Core:

7. Specify and measure behavior when a service is requested by too many clients
8. Identify and repair a performance problem due to sequential bottlenecks
9. Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
10. Identify and repair a performance problem due to communication or data latency
11. Identify and repair a performance problem due to communication or data latency
12. Identify and repair a performance problem due to resource management overhead
13. Identify and repair a reliability or availability problem

PDC-Algorithms

CS Core:

1. Expressing and implementing algorithms (See also FPL-PDC)
    a. 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.
    b. Basic examples: map/reduce
2. Survey of common application domains
   (with reference to the following table)

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

KA Core:

3. Algorithmic Domains. **One of more of**:
    a. Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning
    b. Data processing: sorting, searching and retrieval, concurrent data structures
    c. Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics
    d. Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms
    e. Computational Logic: SAT, concurrent logic programming

f. Graphics and computational geometry: Transforms, rendering, ray-tracing
g. Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms; Real-time constraints; Batching, prioritization, partitioning, Decentralization via work-stealing and related techniques;
h. Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.

Illustrative Learning Outcomes

CS Core:

1. Implement a parallel/distributed component based on a known algorithm
2. Write a data-parallel program that for example computes the average of an array of numbers.
3. 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).
4. Improve the performance of a sequential component by introducing parallelism and/or distribution
5. Choose among different parallel/distributed designs for components of a given system

KA Core:

6. 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.
7. Critique the design and implementation of an existing component or application, or one developed by classmates
8. Compare the performance and energy efficiency of multiple implementations of a similar design; for example multicore versus clustered versus GPU.

Professional Dispositions

- Meticulous: Attention to detail is essential when applying constructs with non-obvious correctness conditions.
- Persistent: Developers must be tolerant of the common need to revise initial approaches when solutions are not self-evident

Math Requirements

- CS Core: Logic, discrete structures including directed graphs.
- KA-Core: Math underlying topics in linear algebra, differential equations

Course Packaging Suggestions

Parallel Computing

- PDC-A: Programs and Execution (4 hours)

- PDC-B: Communication (6 hours)
- PDC-C: Coordination (6 hours)
- PDC-D: Software Engineering (4 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

Distributed Computing

- PDC-A: Programs and Execution (4 hours)
- PDC-B: Communication (3 hours)
- PDC-C: Coordination (3 hours)
- PDC-D: Software Engineering (3 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

High-performance Computing
HPC no prerequisite:

- PDC-A: Programs and Execution (4 hour)
- PDC-B: Communication (6 hours)
- PDC-C: Coordination (6 hours)
- PDC-D: Software Engineering (5 hours)
- PDC-E: Algorithms and Application Domains (11 hours)

HPC with Parallel Computing as prerequisites:

- PDC-A: Programs and Execution (1 hour)
- PDC-B: Communication (2 hours)
- PDC-C: Coordination (2 hours)
- PDC-D: Software Engineering (2 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

More extensive examples and guidance for courses focusing on HPC are provided by the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing (http://tcpp.cs.gsu.edu/curriculum/).

Committee

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

Members:

- Sherif Aly, American University of Cairo, Cairo, Egypt
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Qiao Xiang, Xiamen University, China
- Dan Grossman, University of Washington, Seattle, USA
- Sebastian Burckhardt, Microsoft Research
- Vivek Sarkar, Georgia Tech, Atlanta, USA
- Maurice Herlihy, Brown University, Providence, USA
- Sheikh Ghafoor, Tennessee Tech, USA

- Chip Weems, University of Massachusetts, Amherst, USA

Contributors:

- Paul McKenney, Meta, Beaverton, OR, USA
- Peter Buhr, University of Waterloo, Waterloo, Ontario, Canada