CS3210 Course Summary

Lecture 13

Course Objectives

- Provide an **introduction to the field** of parallel computing with **hands-on** parallel programming experience on **real parallel machines**

- **Four major parts:**
  1. Parallel computing platforms
  2. Parallel computation models
  3. Parallel performance
  4. Parallel computing trends

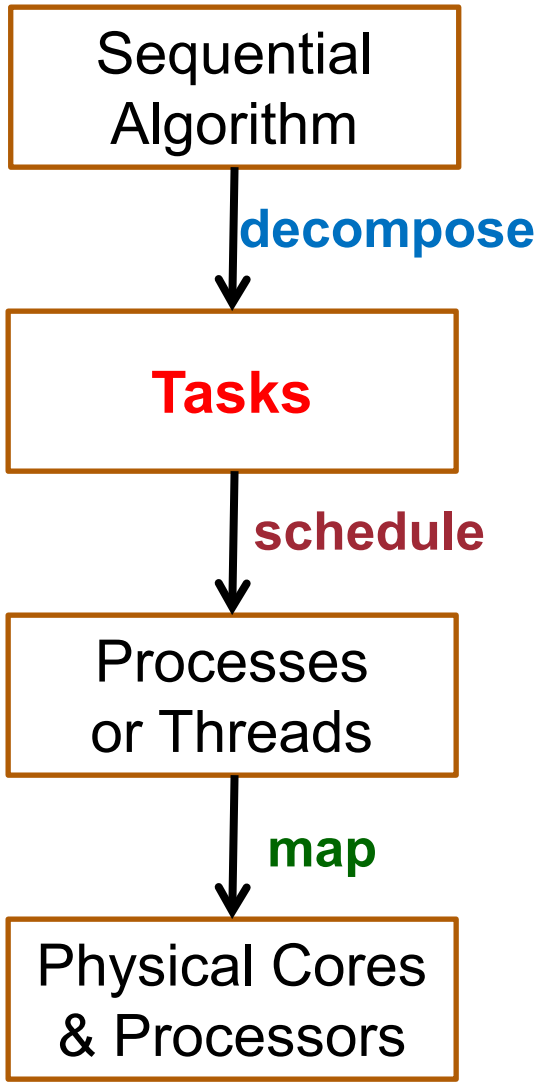
High-level Structure of the Module

- L1-L7: **Shared-memory models**
  - Architecture, memory consistency, programming
  - OpenMP and CUDA
- L8-L11: **Distributed memory models**
  - Architecture, interconnects, programming
  - MPI
- with an emphasis on **Parallel performance**

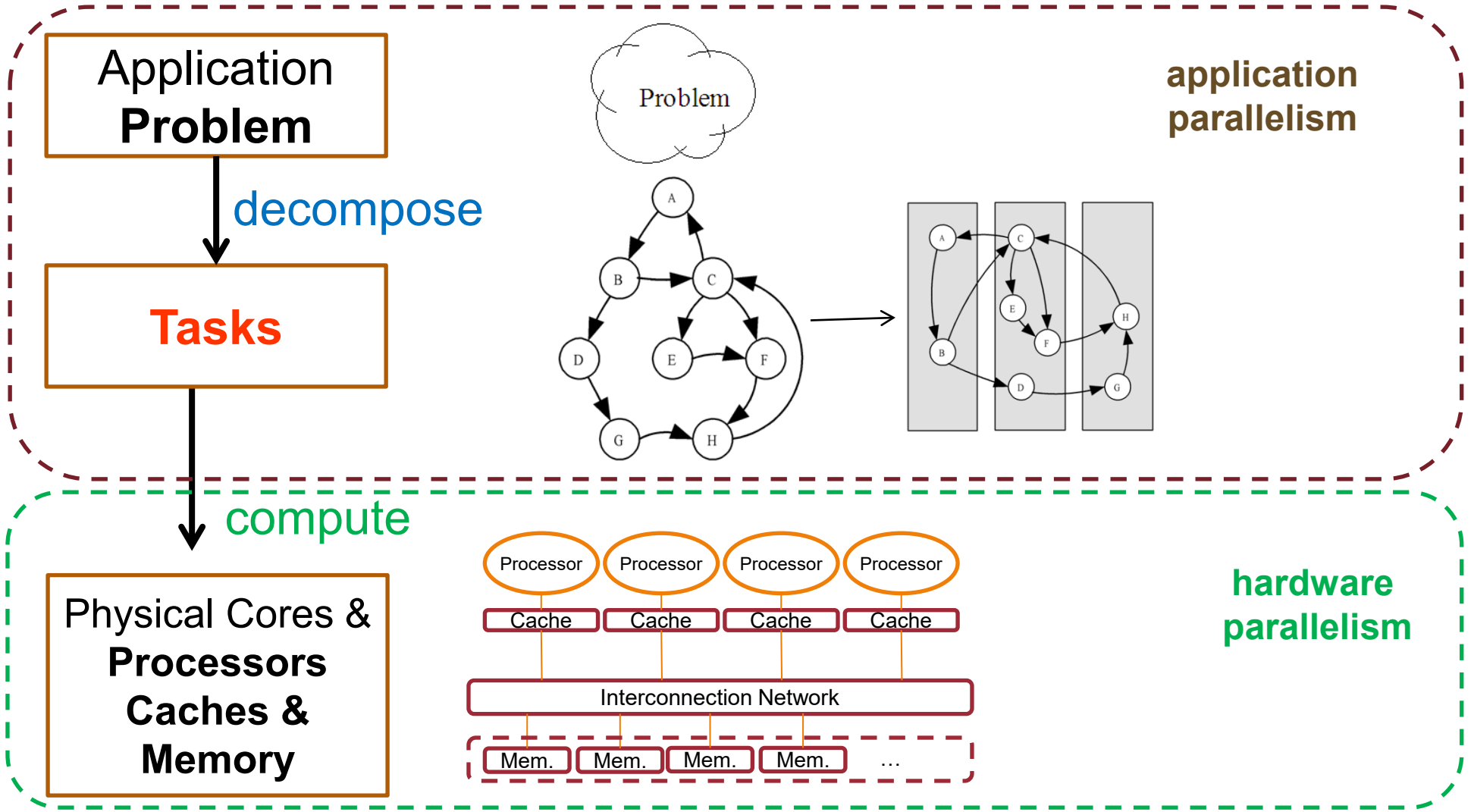
Program Parallelization: Steps

- 3 main steps:

1. **Decomposition** of the computations
2. **Scheduling** (assignment of tasks to processes (or threads))
3. **Mapping** of processes (or threads) to physical processors (or cores)



Parallel Computing

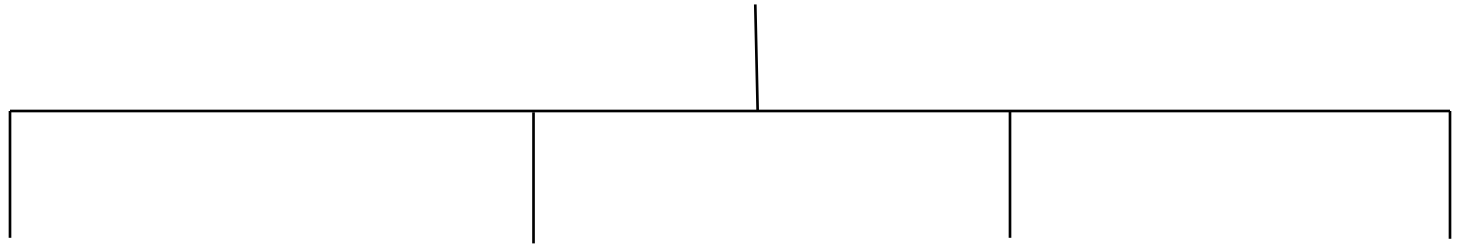


Case Study (CS): Summit Supercomputer

- No. 2 in TOP500
- No. 8 in GREEN500
- At Oak Ridge National Labs
- Partnership
  - IBM
  - Nvidia
- Launched in 2018
- Overall:
  - 2,414,592 cores IBM POWER9
  - 2,801,664 GB memory
  - 27,648 NVIDIA Volta V100s
  - 148,600 TFlop/s Linpack performance
  - 10,096.00 kW

Let's see how CS3210 helps in understanding these details!

PARALLEL COMPUTING PLATFORMS



L03: Parallel Computing Platforms

Parallel Computing Platforms

- Processor Architecture and Technology Trends
  - Various forms of parallelism
- Flynn's Parallel Architecture Taxonomy
- Memory Organization
  - Distributed-memory Systems
  - Shared-memory Systems
  - Hybrid (Distributed-Shared Memory) Systems
- Architecture of Multicore Processors

Source of Processor Performance Gain

- **Parallelism of various forms** are the main source of performance gain

- Parallelism at the:
  - **Bit Level**
  - **Instruction Level**
  - **Thread Level**
  - **Process Level**
  - **Processor Level:**
    - Shared Memory
    - Distributed Memory

Single Processor

Multiple Processors

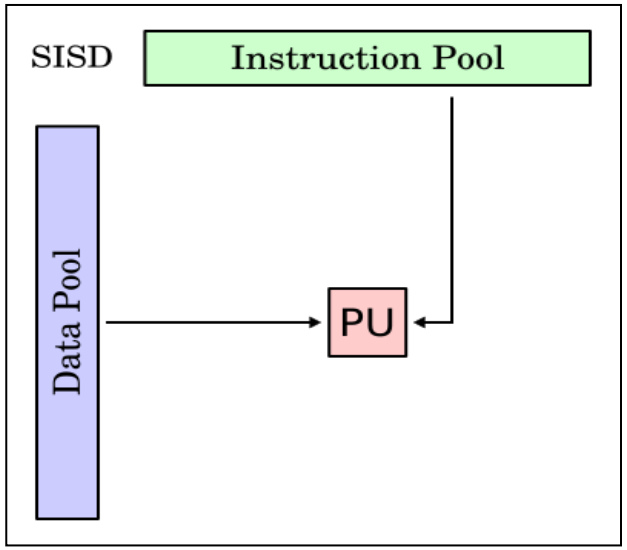
Flynn's Parallel Architecture Taxonomy

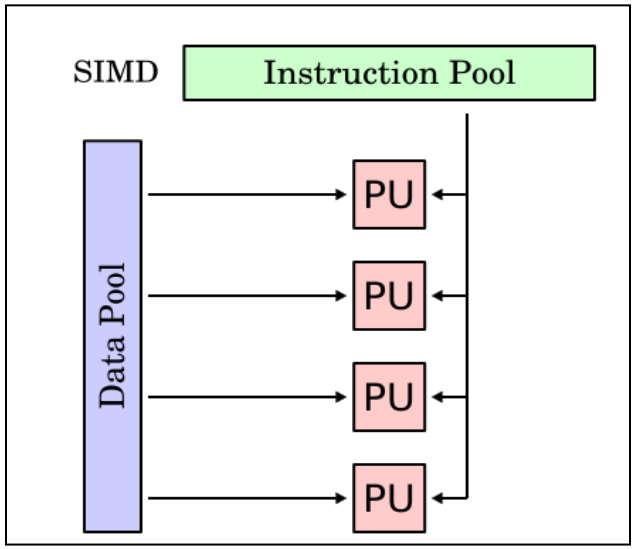
- One commonly used taxonomy of parallel architecture:
  - Proposed by **M. Flynn** in 1972
  - Based on the parallelism of **instruction** and **data** streams in the **most constrained** component of the processor

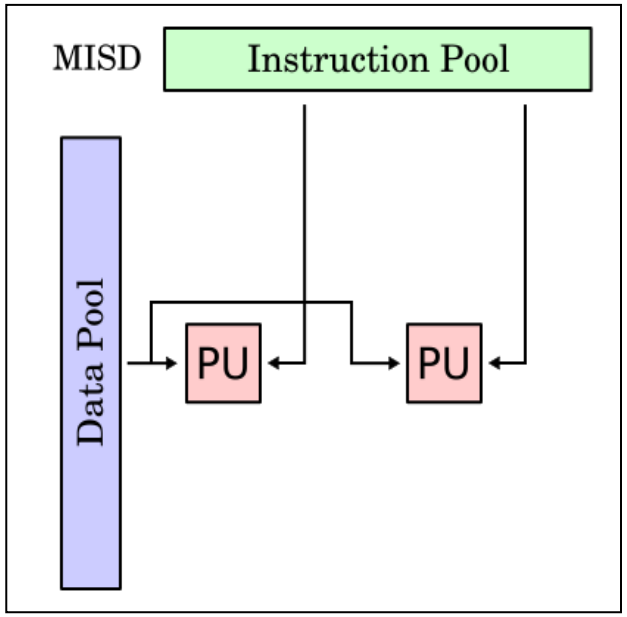
- **Instruction stream:**
  - A single execution flow (a single PC)

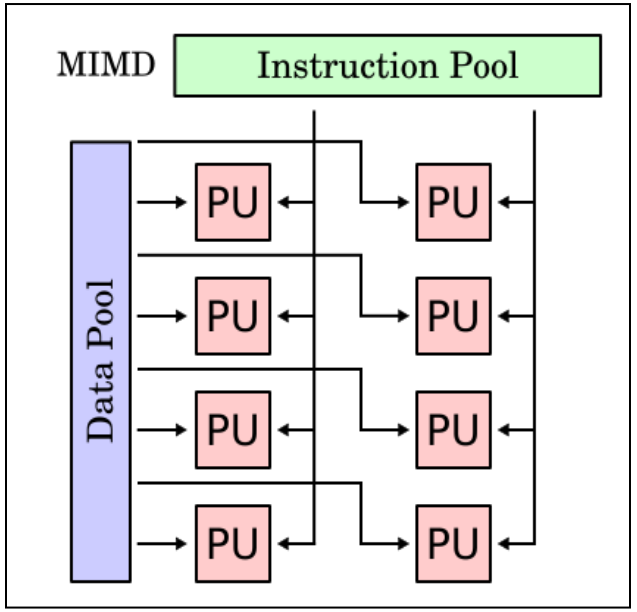
- **Data stream:**
  - Data being manipulated by the instruction stream

S/M Instruction x S/M Data

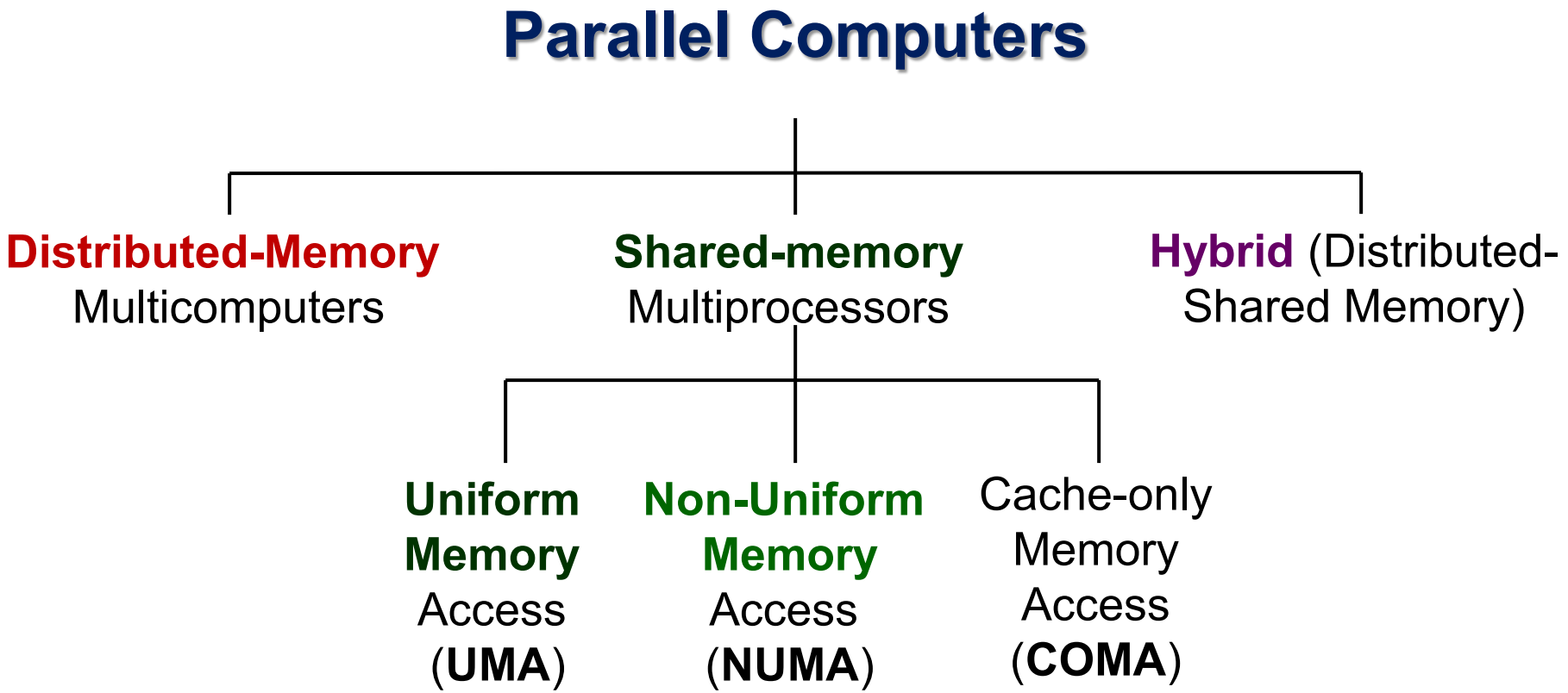




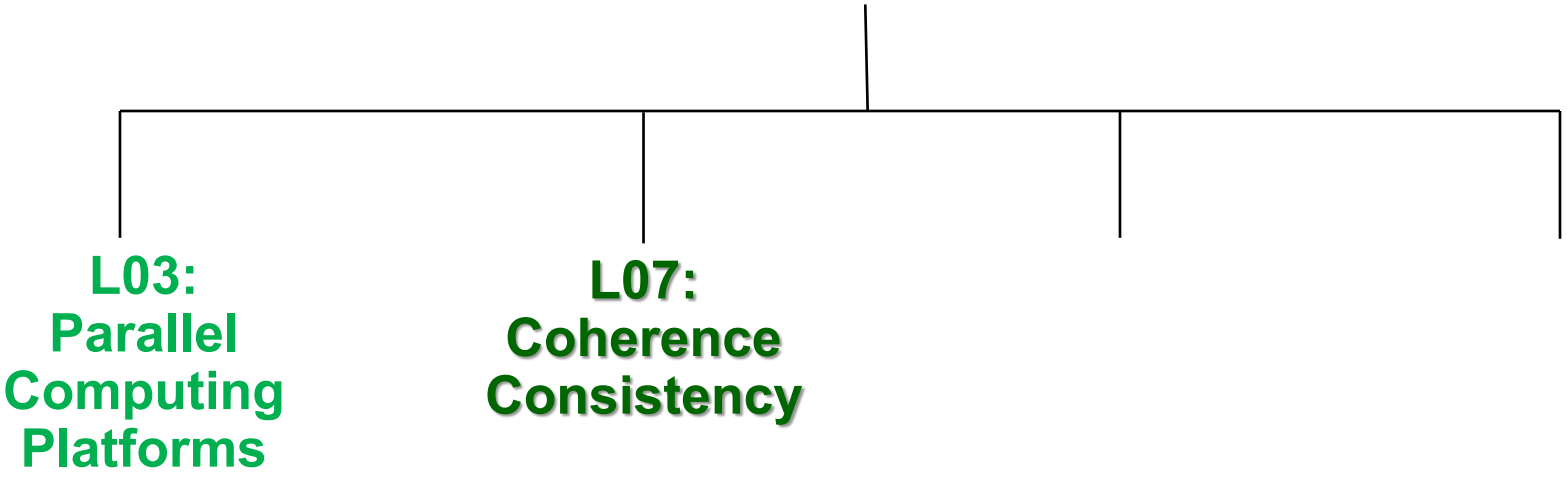




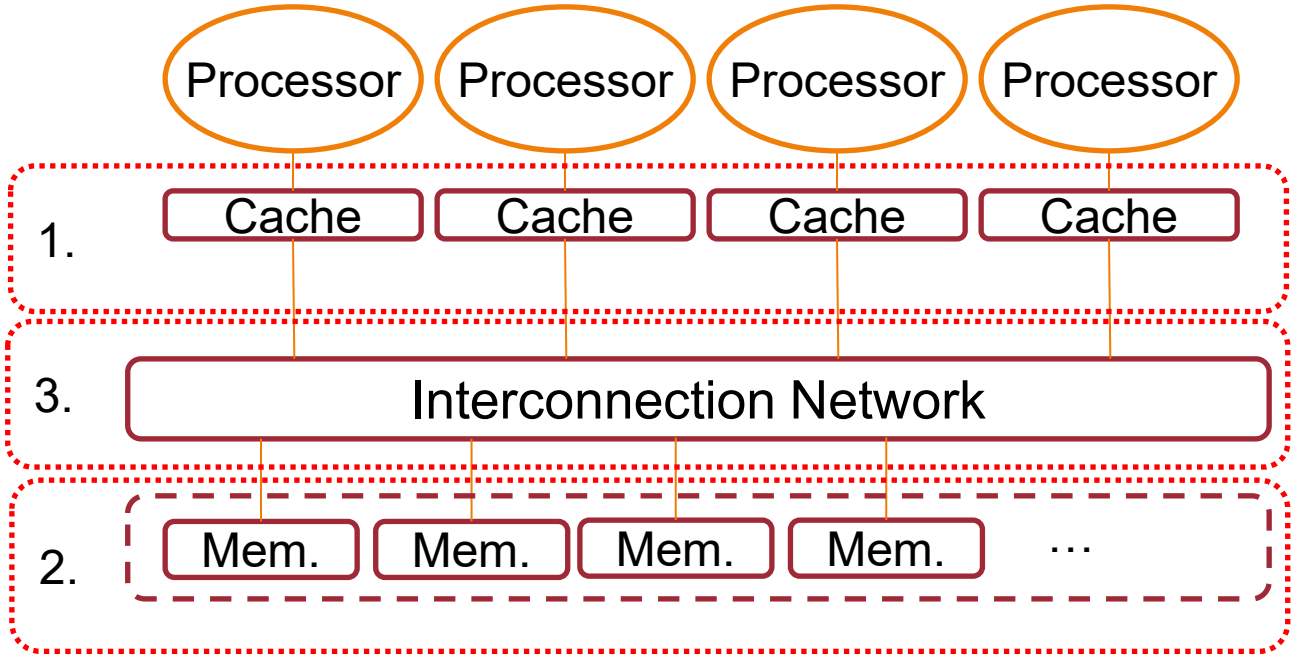
Memory Organization of Parallel Computers



PARALLEL COMPUTING PLATFORMS



1. Cache Coherence
2. Memory Consistency
3. Interconnection Networks



Cache Coherence and Memory Consistency

- Cache coherence ensures that each processor has consistent view of memory through its local cache
  - But does not specify in which order write access becomes visible to other processors
  - For the same address
- Memory consistency models define
  - Constraints on the order in which memory operations can appear to execute
  - For multiple addresses
- The model is then used:
  - By programmers to reason about correctness
  - By system designers to decide the reordering possible by hardware and compiler

Sequential Consistency Model (SC)

- A sequentially consistent memory system:
  - Every processor issues its memory operations in program order

- Extension of uniprocessor memory model:
  - Intuitive but can result in loss of performance

Write-to-Read Program Order

- **Key Idea:**
  - Allow a read on processor **P** to be reordered w.r.t. to the previous write of the same processor
  - Different timing of the return of the read defines different models

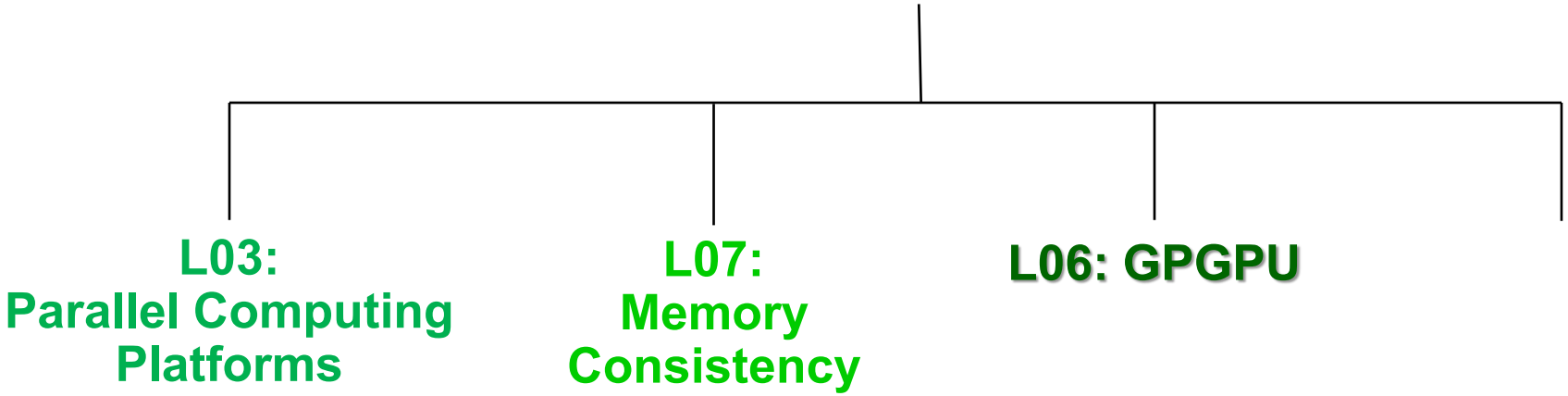
- Example Models:
  - **T**otal **S**tore **O**rdering (**TSO**) :
    - Return the value written by P earlier without waiting for it to be serialized
  - **P**rocessor **C**onsistency (**PC**) :
    - Return the value of any write (even from another processor) before the write is propagated or serialized

Write-to-Write Program Order

- **Key Idea:**
  - Writes can bypass earlier writes (to different locations) in write buffer
  - Allow **write miss** to overlaps and hide latency

- Example Model:
  - **P**artial **S**tore **O**rdering (**PSO**)
    - Relax **W → R** order
      - Similar to **TSO**
    - Relax **W → W** order
    - **Guarantees write atomicity**

PARALLEL COMPUTING PLATFORMS



GPGPU

- CUDA Programming Model
  - GPU Architecture
  - Thread Hierarchy
  - Memory model
- CUDA Software Development
- Programming in CUDA
- Optimizing CUDA Programs

CUDA

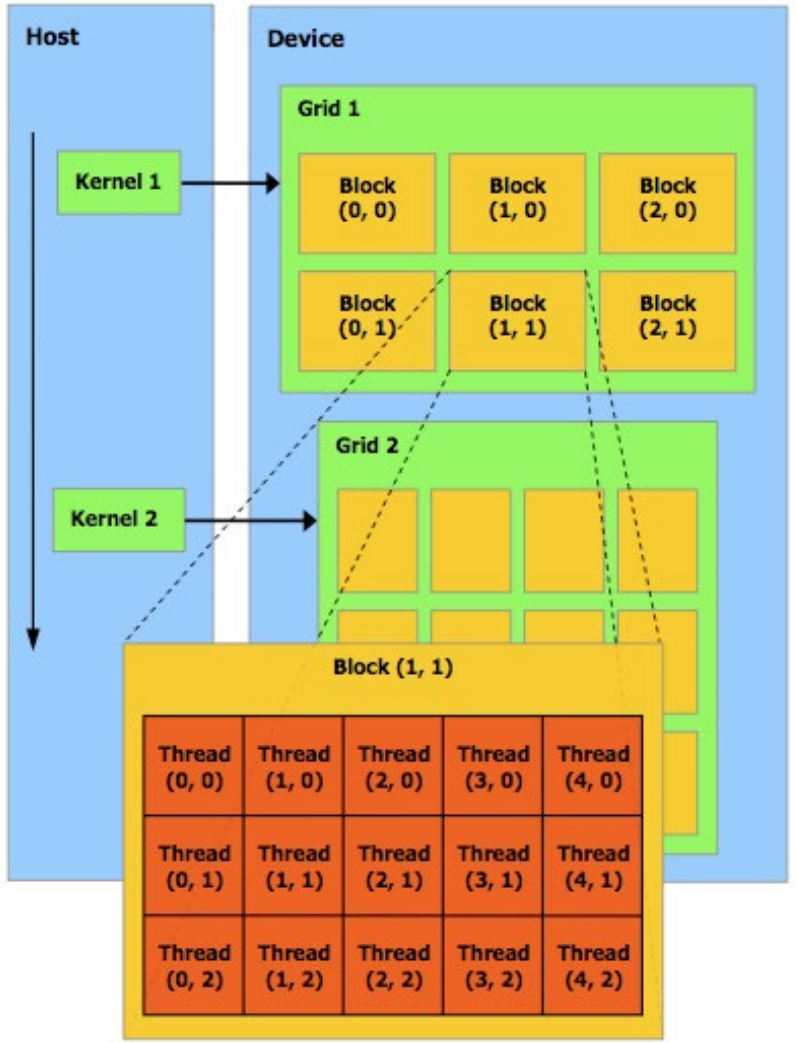
- “Compute Unified Device Architecture”
- Massively hardware multithreaded
  - GPU = dedicated many-core co-processor
- General purpose programming model
  - Simple extension to standard C
  - Mature software stack (high-level and low-level access)
  - User launches batches of threads on the GPU

GPU Architecture

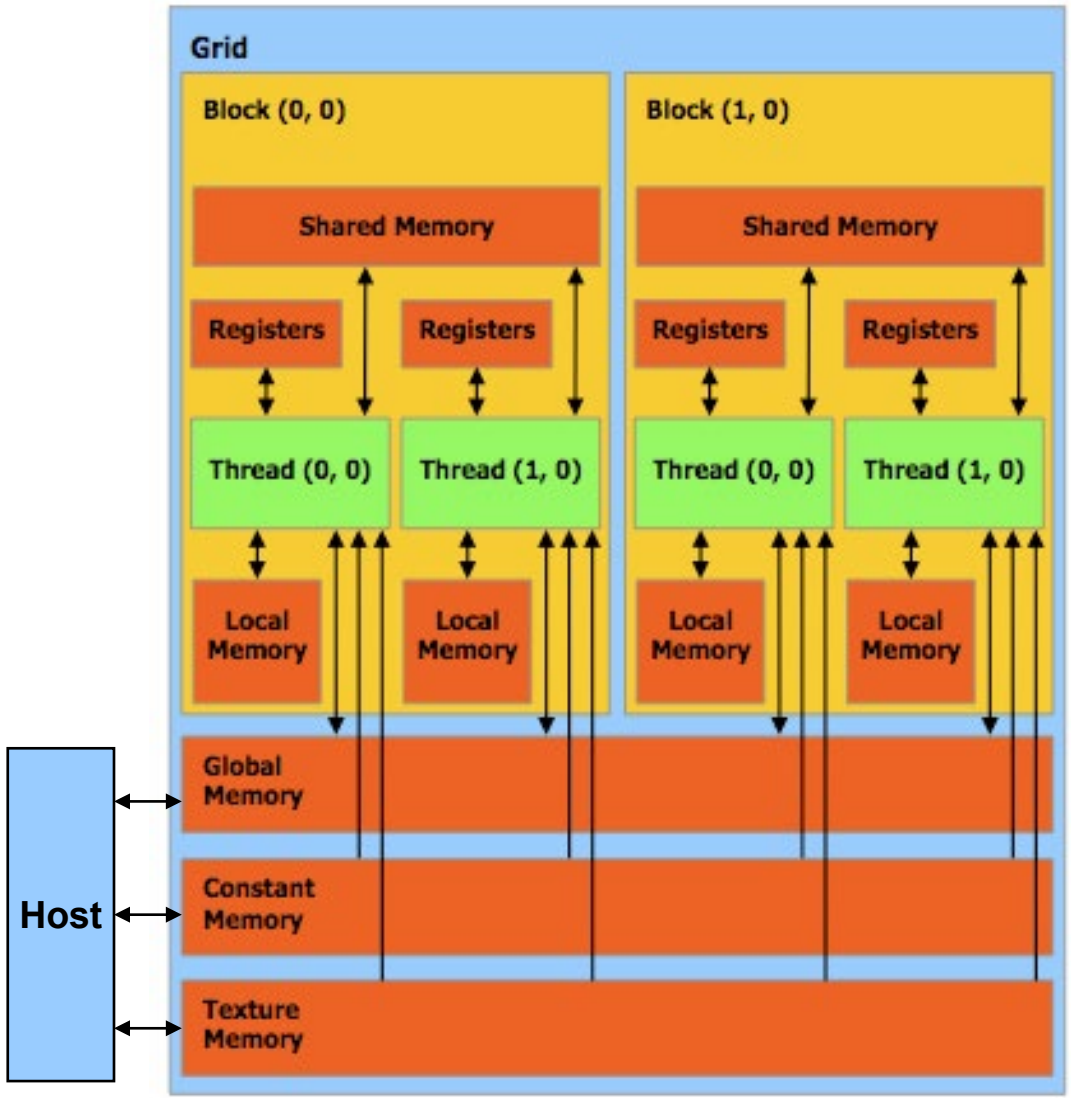
- Multiple Streaming Multiprocessors (SMs)
  - Memory and cache
  - Connecting interface (usually PCI Express)
- SM consists of multiple compute cores
  - Memories (registers, L1 cache, texture memory, and shared memory)
  - Logic for thread and instruction management

CUDA Thread Hierarchy

- A kernel is executed by a grid of thread blocks
- Block: batch of threads that can
  - Share data through shared memory
  - Synchronize their execution
- Threads from different blocks cannot cooperate



CUDA Memory Model



Overall Optimization Strategies

(1) Maximizing parallel execution

- Restructure algorithm to expose as much **data parallelism** as possible
- Map to hardware by carefully choosing the execution configuration of each kernel invocation

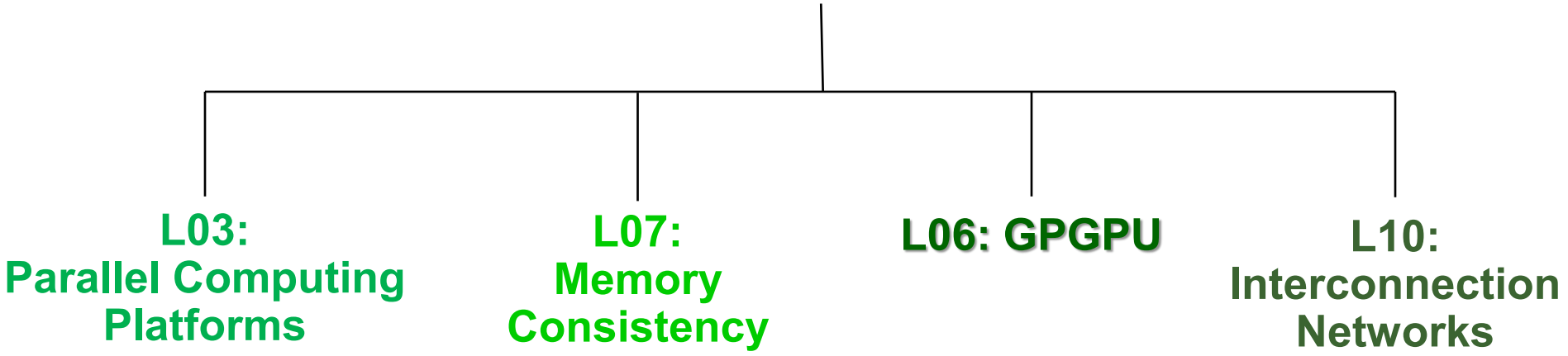
(2) Optimizing memory usage to achieve maximum memory bandwidth

- Different memory spaces and access patterns have vastly different performance

(3) Optimizing instruction usage to achieve maximum instruction throughput

- Use high throughput arithmetic instructions
- Avoid different execution paths within same warp

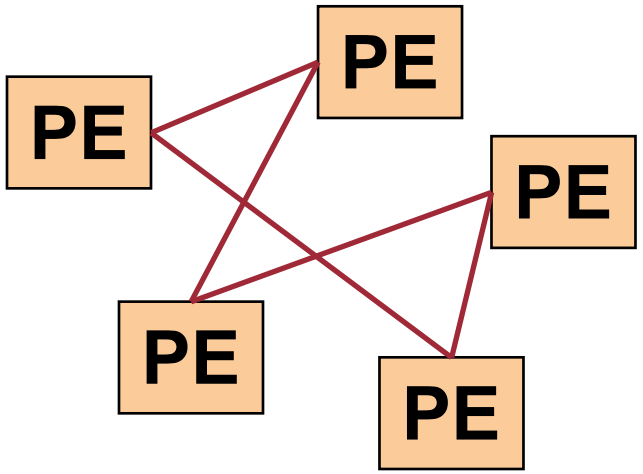
PARALLEL COMPUTING PLATFORMS

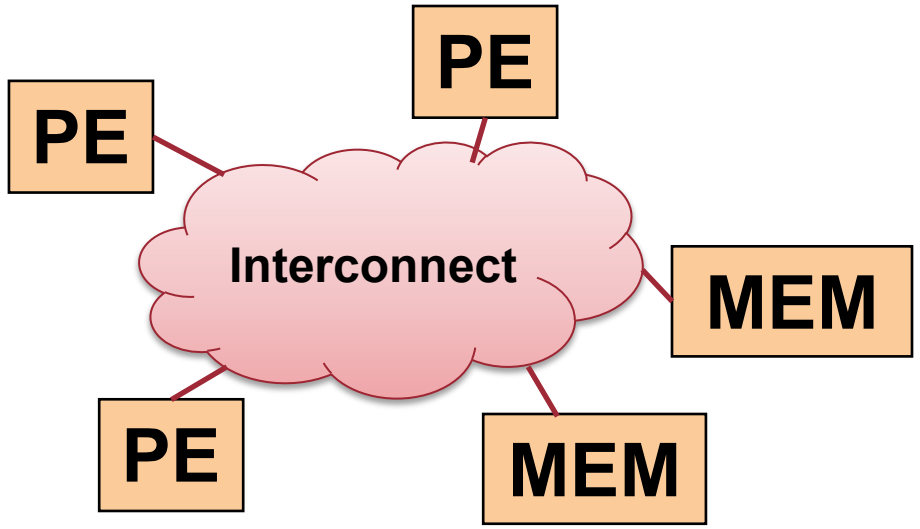


Interconnection Networks

- **Motivating Examples**
  - Issues
- **Topology**
  - Metrics
  - Direct and Indirect Interconnection Networks
- **Routing**
  - Example algorithms

Topology: Major Type





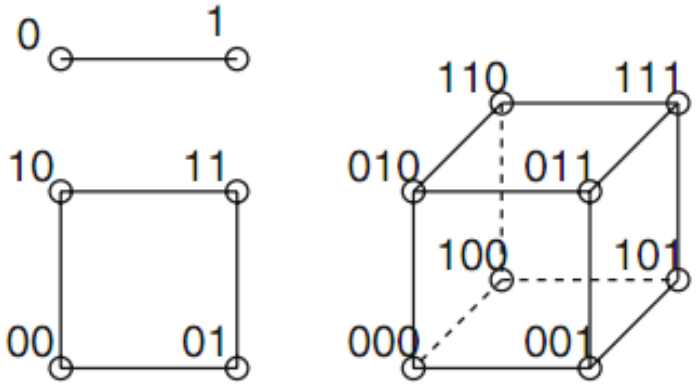
Direct Interconnection

- Also known as **Static** or **Point-to-Point**
- Each endpoint is a PE

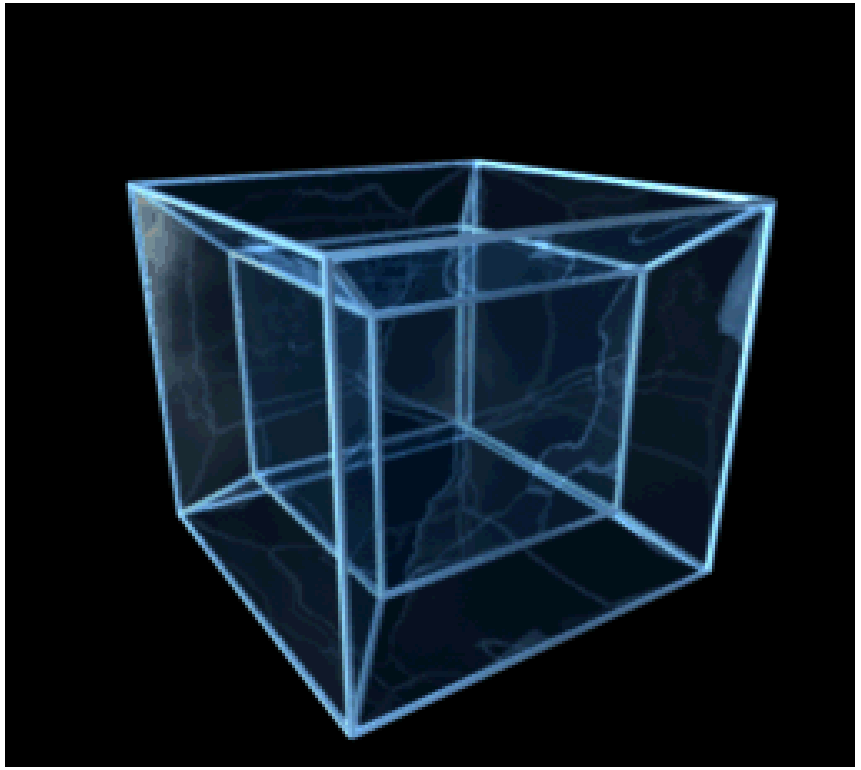
Indirect Interconnection

- Also known as Dynamic
- Interconnect is formed by switches

Hypercube



$$n = 2^k$$

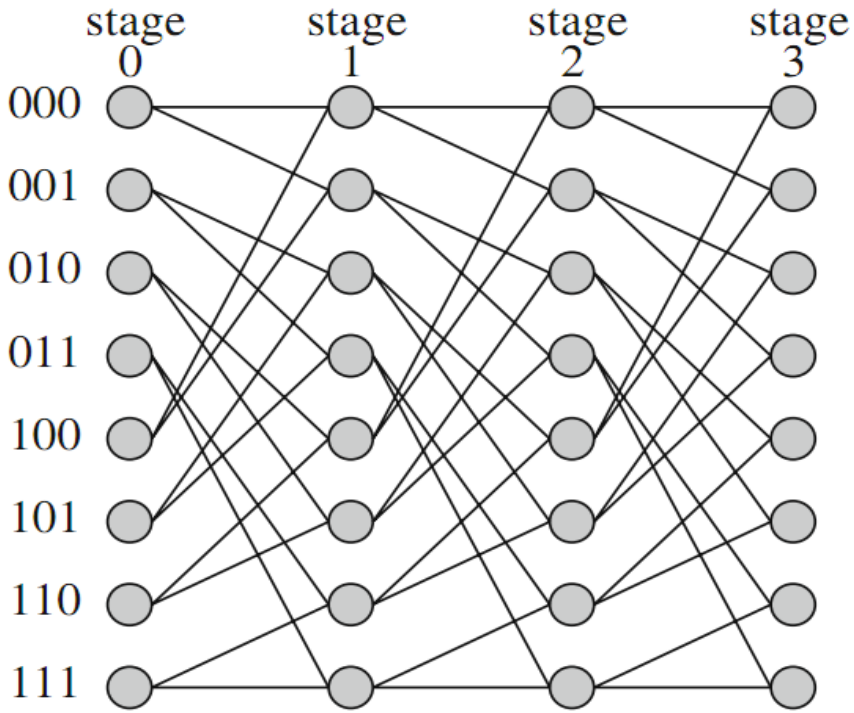


k-dimensional hypercube for k=1,2,3,4

Degree **g(G)**	**Log n**
Diameter **δ (G)**	**Log n**
Edge Connectivity **ec(G)**	**Log n**
Bisection Width **B(G)**	**n/2**

Omega Network

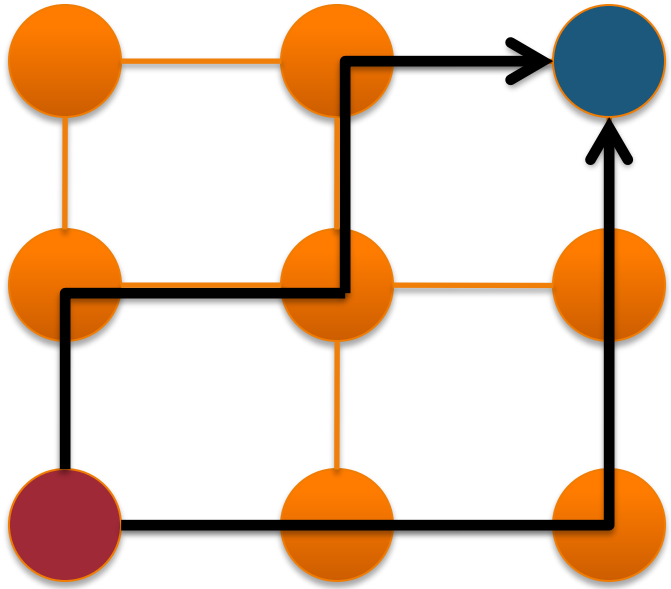
16 × 16 Omega Network



- One unique path for every input to output
- An **n × n** Omega network has **log n stages**
  - **n/2 switches per stage**
  - Connections between stages are regular
  - aka **(log n – 1) – dimension Omega Network**

Routing Overview

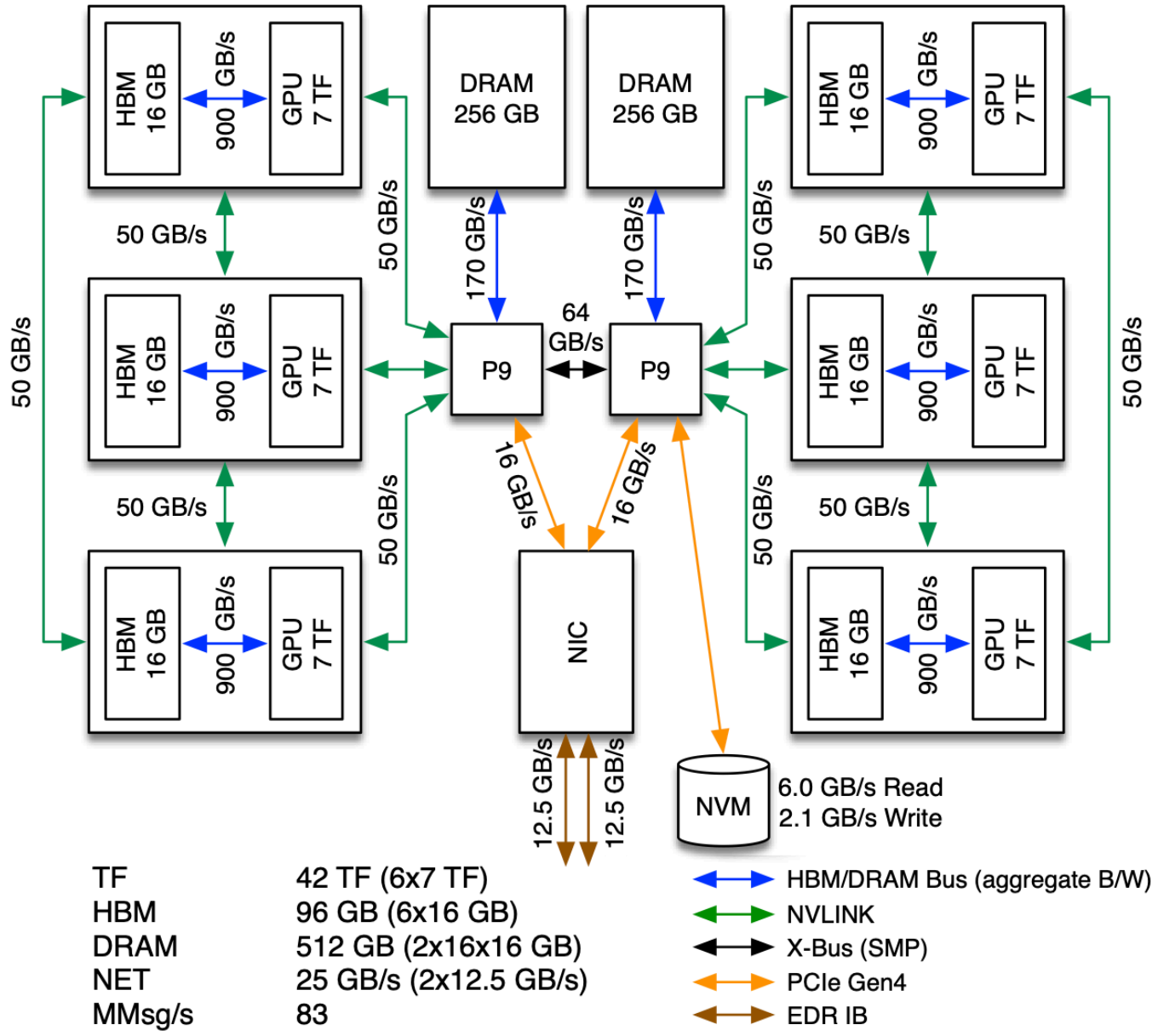
- Routing algorithm determines path(s) from source to destination
  - Within a given interconnection topology



CS: Deconstructing Summit

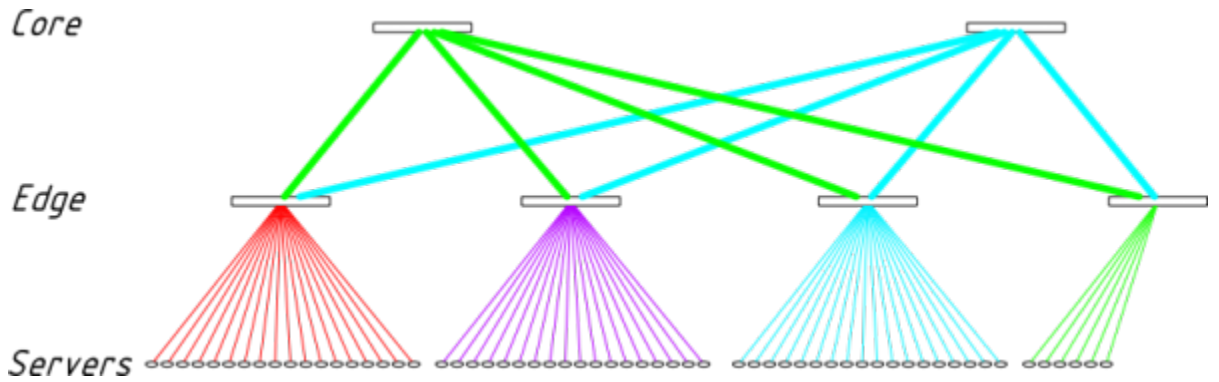
- 4,608 nodes:
  - water-cooled
  - 2 * 22 cores IBM Power9 CPUs 3.07GHz
    - 4 slices to provide SMT1, 2, 4
  - 6 NVIDIA Volta V100 GPUs
  - 512GB of coherent DDR4 and 96GB HBM2 (High Bandwidth Memory)
  - 1,600GB of non-volatile RAM
  - 96 lanes of PCIe 4.0
  - dual-port Mellanox EDR InfiniBand adapter

1 Node in Summit

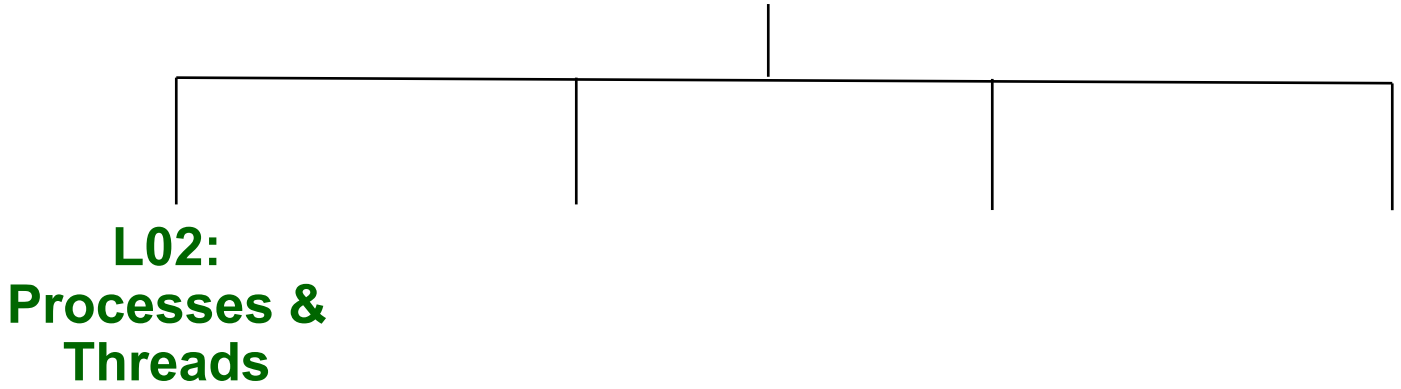


CS: Interconnect in Summit

- Non-blocking Fat Tree topology
  - Dual-rail EDR InfiniBand network
  - Bandwidth of 23 GB/s



PARALLEL PROGRAMMING MODELS



Processes & Threads

- Processes
- Threads
- Synchronization
  - Race condition
  - Critical section
  - Deadlock
  - Starvation

Processes

- A program in execution
  - Comprises:
    - executable program (PC),
    - global data
      - OS resources: open files, network connections
    - stack or heap
    - current values of the registers (GPRs and Special)
  - Own address space ➔ exclusive access to its data
  - Two or more processes exchange data ➔ need explicit communication

Disadvantages

- Creating a new process is costly
  - Overhead of system calls
  - All data structures must be allocated, initialized and copied
- Communicating between processes costly
  - Communication goes through the OS

Threads

- Extension of process model:
  - A process may consist of multiple independent control flows called **threads**
  - The thread defines a sequential execution stream within a process(PC, SP, registers)
- Threads share the address space of the process:
  - All threads belonging to the same process see the same value ➔ **shared-memory architecture**

Synchronization

- Threads cooperate in multithreaded programs
  - Share resources, access shared data structures
  - Coordinate their execution
    - One thread executes relative to another
- For correctness, control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- Use synchronization
  - Restrict the possible interleaving of thread executions

*Discuss in terms of threads, also applies to processes

Shared Resources

- Coordinating access to shared resources
  - Basic problem:
    - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/ modified/ written by those threads, then access to the variable must be controlled to avoid erroneous behaviour
  - Mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.

Race Condition

- Two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  - Known as a race condition
- Control access to these shared resources
- Necessary to synchronize access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

Mutual Exclusion

- Use mutual exclusion to synchronize access to shared resources
  - This allows us to have large atomic blocks
- Code sequence that uses mutual exclusion is called critical section
  - Only one thread at a time can execute in the critical section
  - All other threads have to wait on entry
  - When a thread leaves a critical section, another can enter

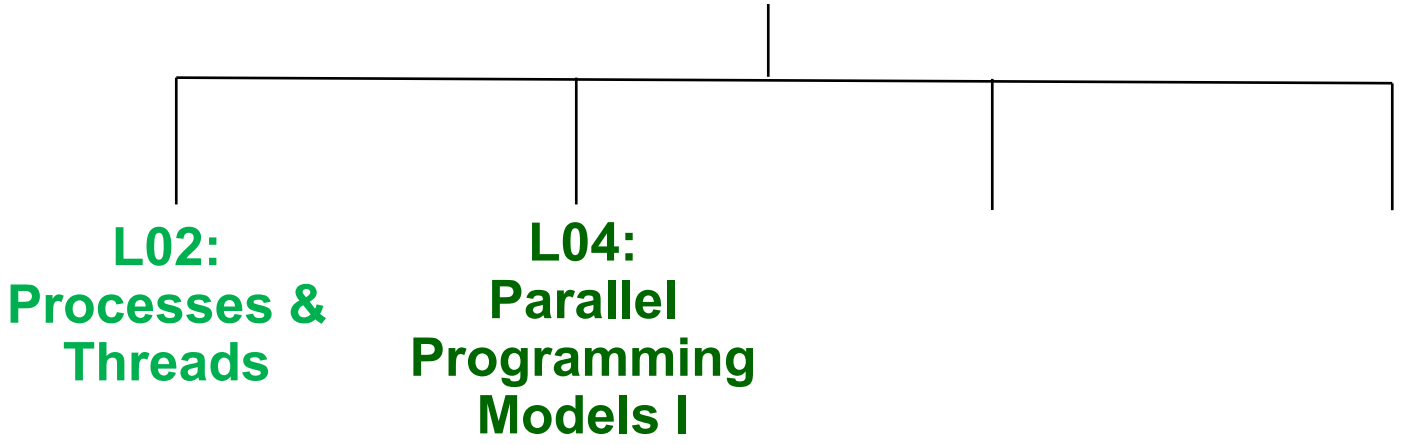
Deadlock

- Definition:
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

Starvation

- Starvation is a situation where a process is prevented from making progress because some other process has the resource it requires
- Starvation is a side effect of the scheduling algorithm
  - OS: A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

PARALLEL PROGRAMMING MODELS



Parallel Programming Models I

- **Program Parallelization**

- **Parallelism**
  - Types of Parallelism
  - Representation of Parallelism
  - Parallel Programming Patterns

Parallel Programming Models

- Provides **mechanism** for the programmers

Level of parallelism
- Instruction level, Statement level, Procedural level, or Parallel loops

Specification of parallelism
- Implicit or user-defined explicit

Execution
- e.g., SIMD or SPMD, synchronous or asynchronous

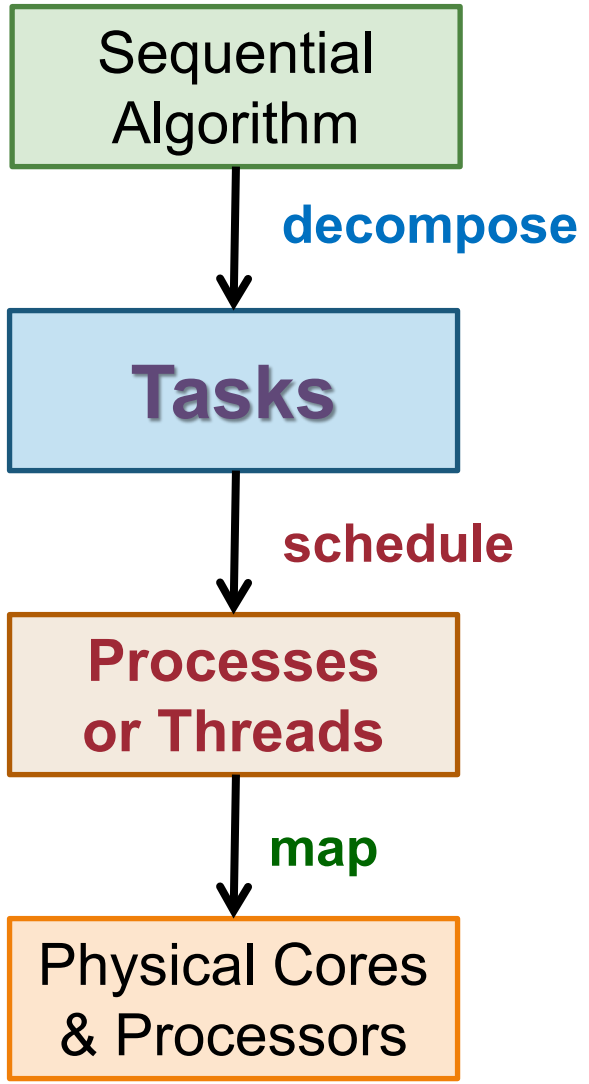
Communication mode
- i.e., message-passing or shared variables

Synchronization (coordination) mechanisms

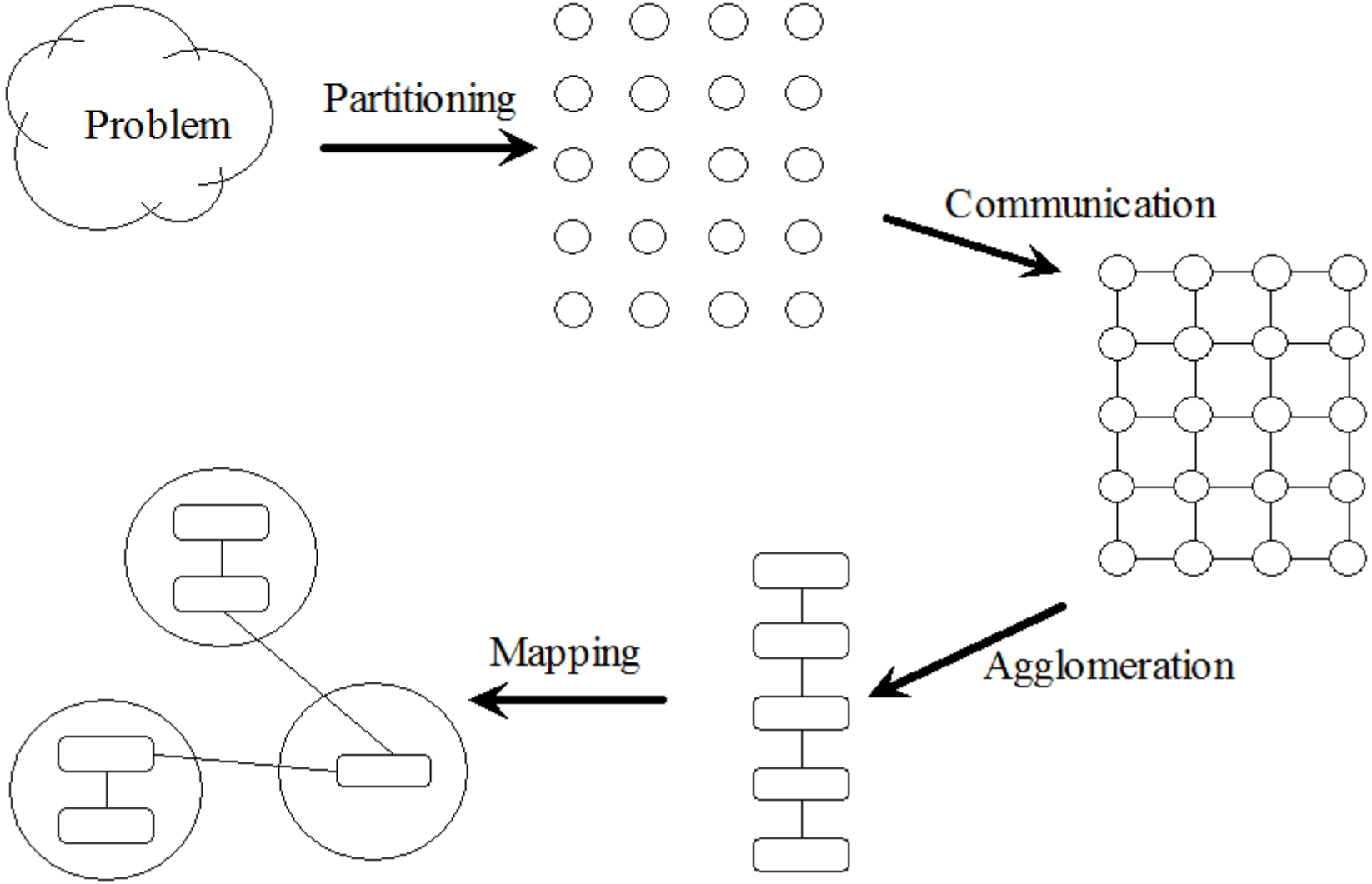
Program Parallelization: Steps

- 3 main steps:

1. **Decomposition** of the computations
2. **Scheduling** (assignment of tasks to processes (or threads))
3. **Mapping** of processes (or threads) to physical processors (or cores)



Foster's Methodology

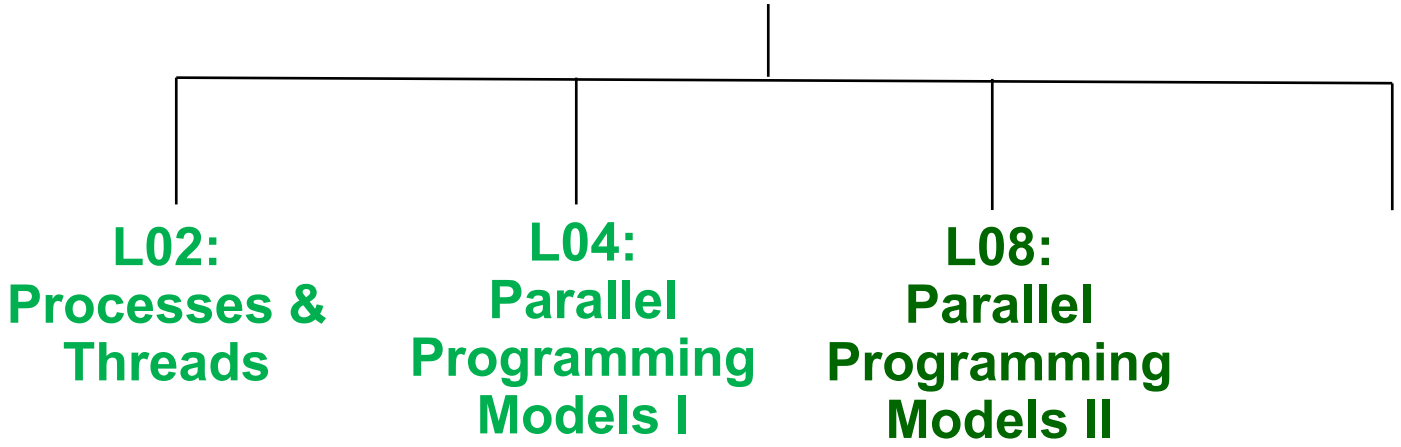


Parallel Programming Patterns

- Provides a coordination structure for tasks:
  - Similar to design patterns from Software Engineering

- **Examples**
  - Fork–Join
  - Parbegin-Parend
  - SPMD and SIMD
  - Master-Slave (Worker)
  - Client-Server
  - Pipelining
  - Task pool
  - Producer-consumer

PARALLEL PROGRAMMING MODELS



Parallel Programming Models II

- **Data Distribution**
  - 1D array
  - 2D array

- **Information Exchange**
  - Shared variables
  - Communication operations

Data distribution for 1D Arrays

- Assumptions:
  - **p** identical processors, $\mathbf{P_1}$, $\mathbf{P_2}$, .., $\mathbf{P_p}$, and with processor rank **i ε {1, 2, .., p}**
  - Array elements numbered from **1** to **n**

- Common distribution patterns:
  - **Blockwise data distribution**
  - **Cyclic data distribution**

Data distribution for 2D Arrays

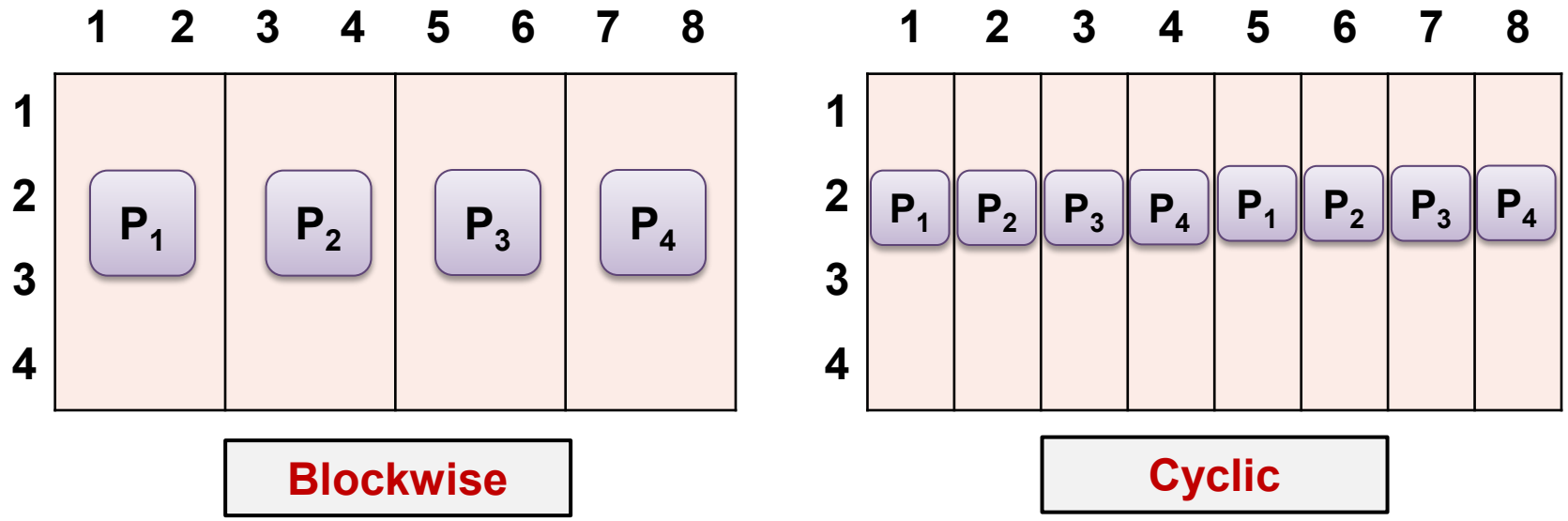
- Combination of **blockwise** / **cyclic** distribution in one or both dimensions can be used

- **One dimension distributions**
  - Along row or column dimensions

- **2D Checkerboard distribution**:
  - Blockwise
  - Cyclic
  - Block-Cyclic

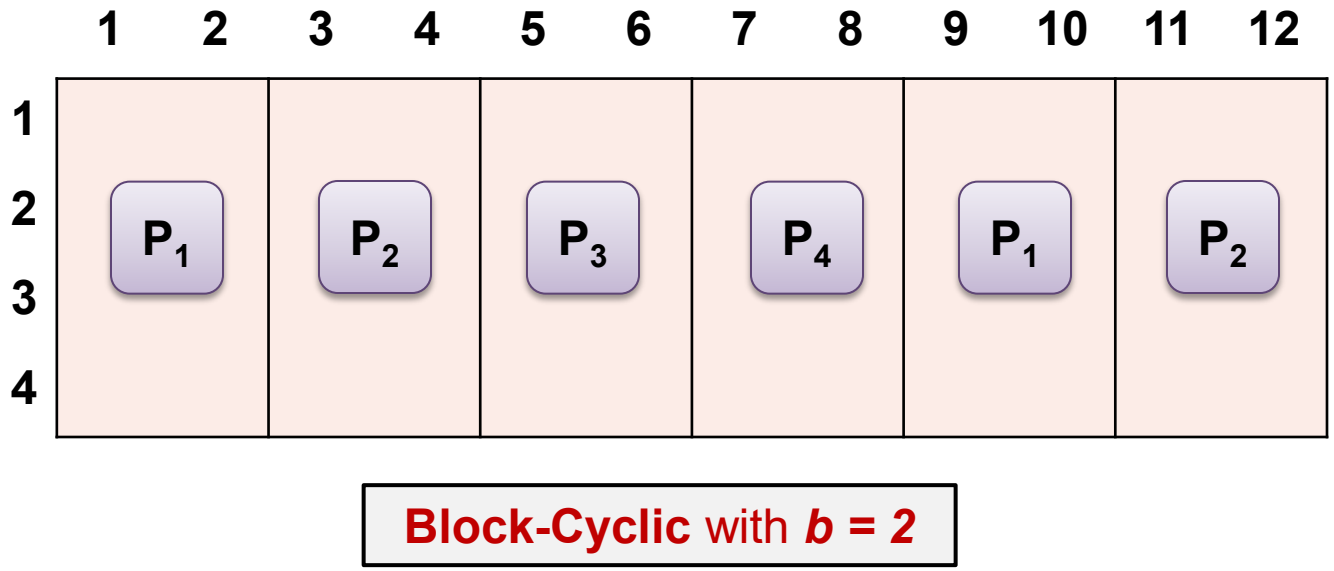
Data distribution for 2D Arrays

- Combination of blockwise / cyclic distribution in one or both dimensions can be used
- One dimension distributions
  - Use the **column dimension** for illustration:

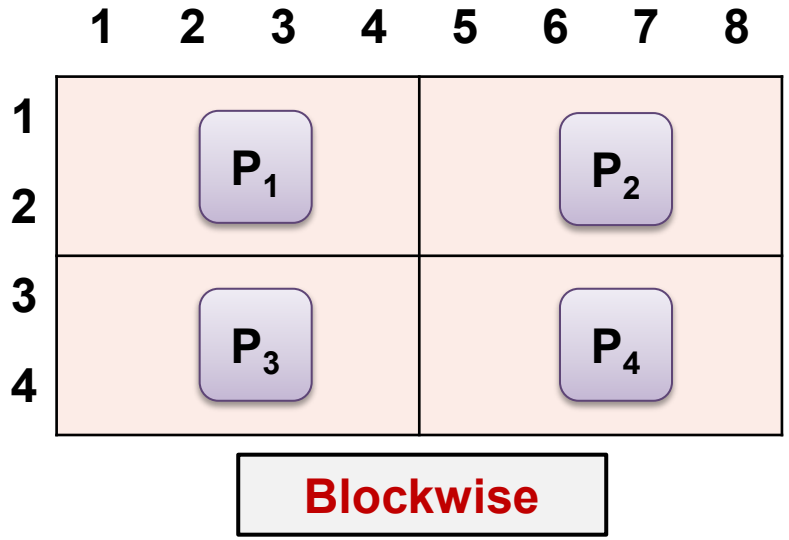


Data distribution for 2D Arrays

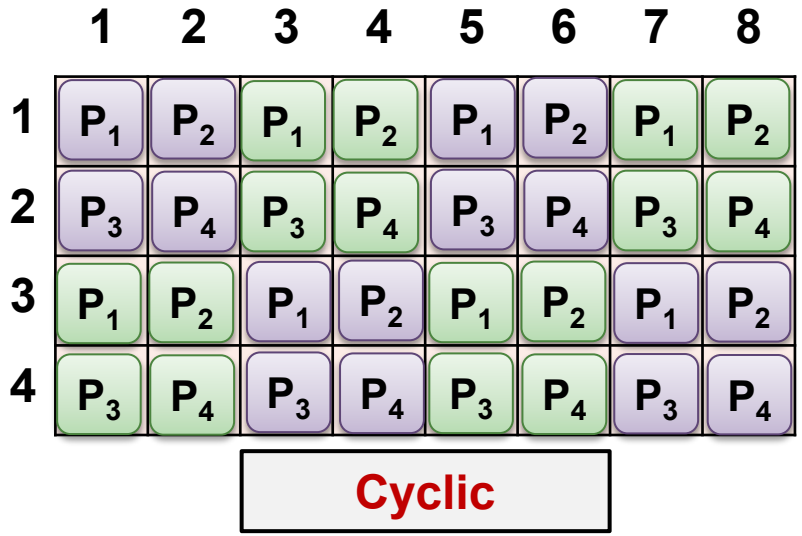
- One dimension distributions
  - **Block-Cyclic** is a new distribution pattern
  - Form blocks of size **b**, then perform cyclic (round robin) allocation



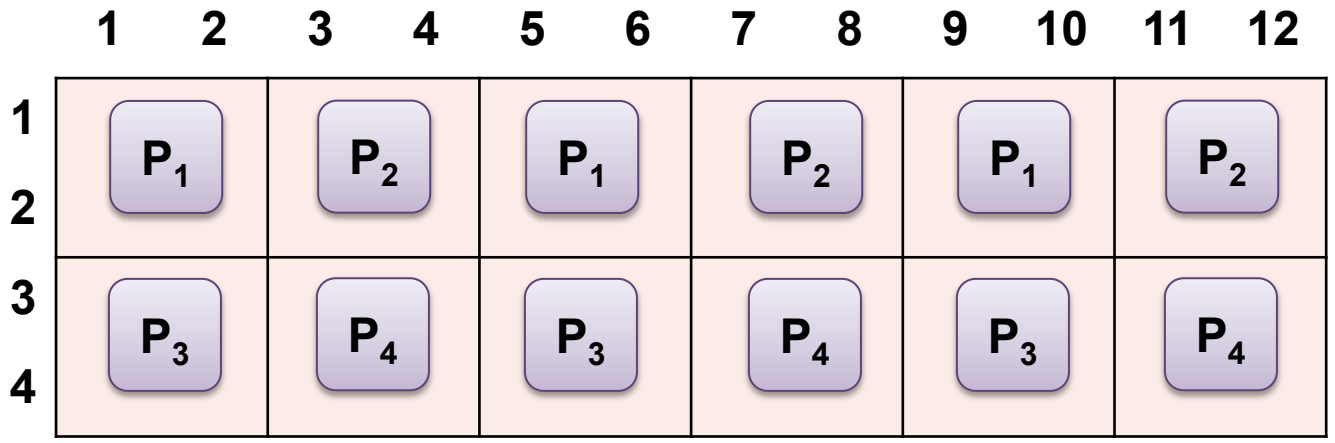
Checkerboard Distribution



Blockwise



Cyclic



Block-Cyclic with $b_1 = 2$, $b_2 = 2$

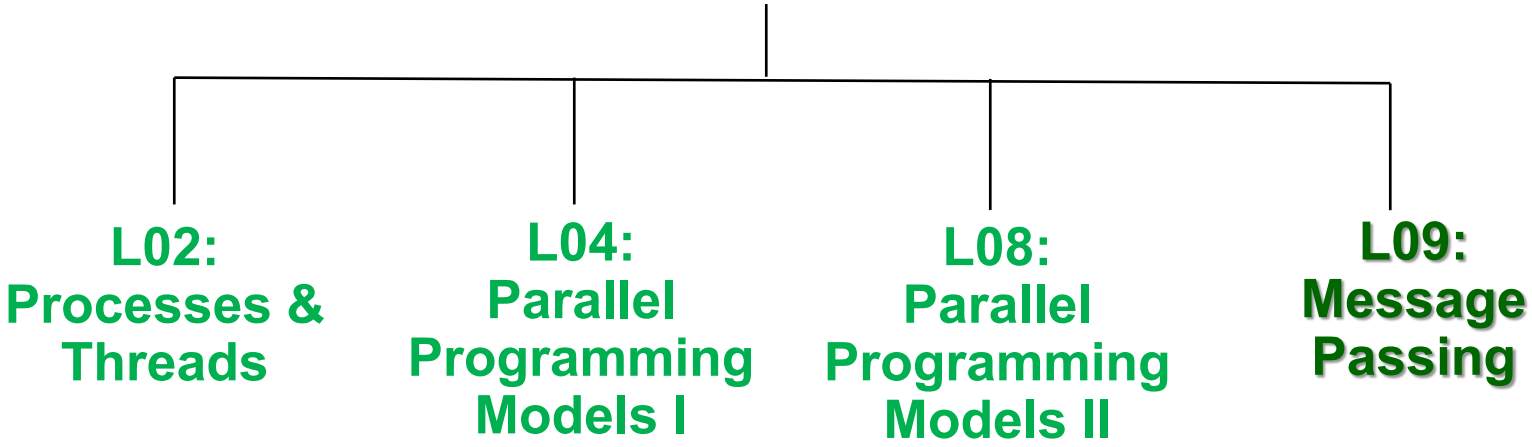
Information Exchange

- **Purpose**
  - Information exchange between the executing processors is necessary for controlling the coordination of different parts of a parallel program execution

- **Shared address space**
  - use **Shared variables**

- **Distributed address space**
  - use **Communication operations**

PARALLEL PROGRAMMING MODELS



Message Passing

- **Message Passing Overview**
- **Communication Building Blocks**
  - Send and Receive Operations
  - Send and Receive Protocols
- **MPI**
  - Initialization, Finalization and Abort
  - Process Groups and Communicators
  - Point-to-point Communication
  - Collective Communication

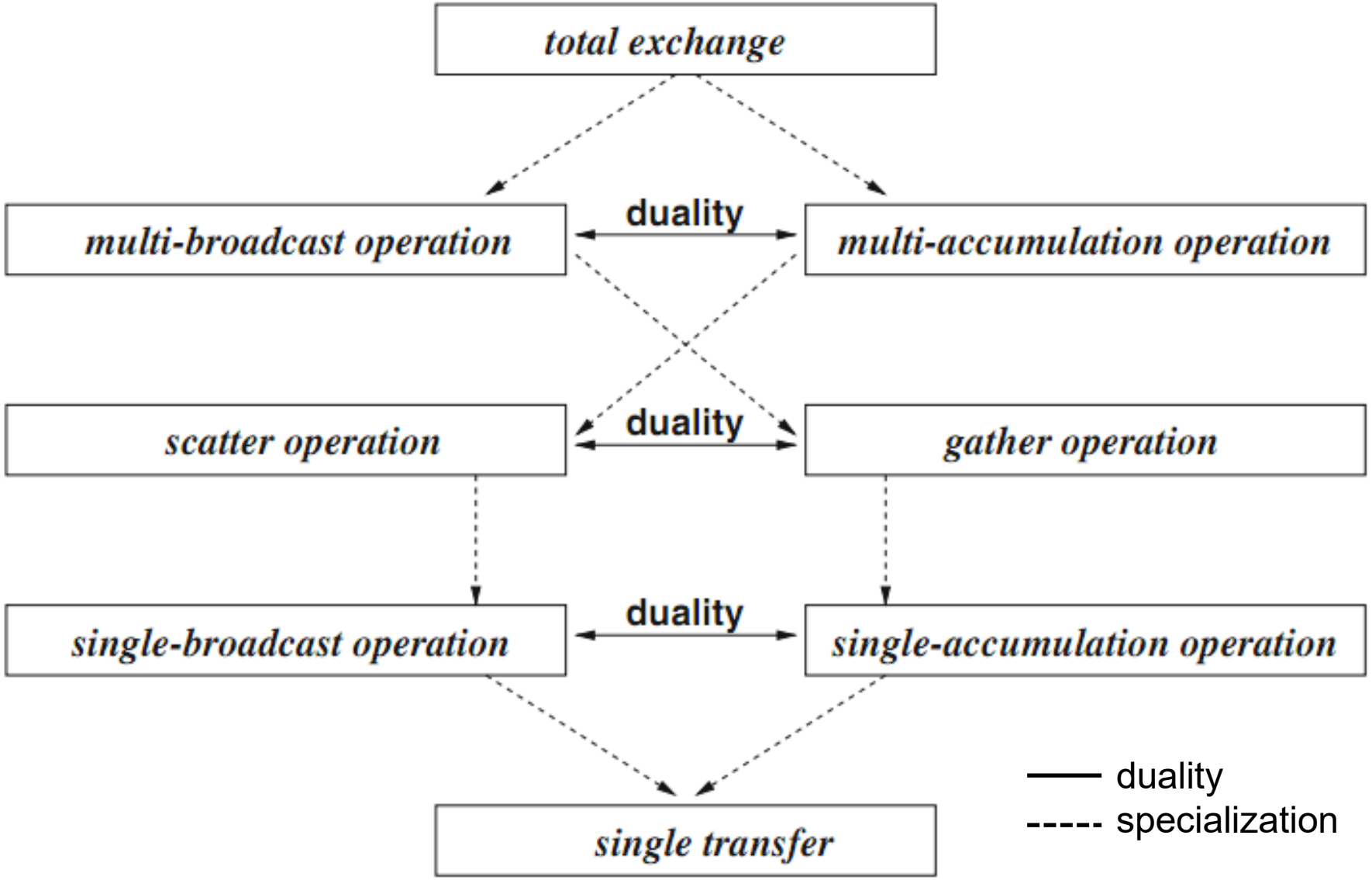
MPI Coverage Outline

- Initialization, Finalization and Abort
- Process Groups and Communicators
- Point-to-point Communication
- Collective Communication

Semantic of MPI Operations

Local view	Global view
Blocking Return from a library call indicates the user is allowed to reuse resources specified in the call	**Synchronous** Communication operation does not complete before both processes have started their communication operation
Non-blocking A procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call	**Asynchronous** Sender can execute its communication operation without any coordination with the receiver

Message Communication Operations



CS: Programming Summit

Compilers

- XL: IBM XL
- LLVM: LLVM compiler infrastructure
- PGI: Portland Group compiler suite
- GNU: GNU Compiler Collection
- NVCC: CUDA C compiler
- MPI: IBM Spectrum MPI
- OpenMP

Running jobs

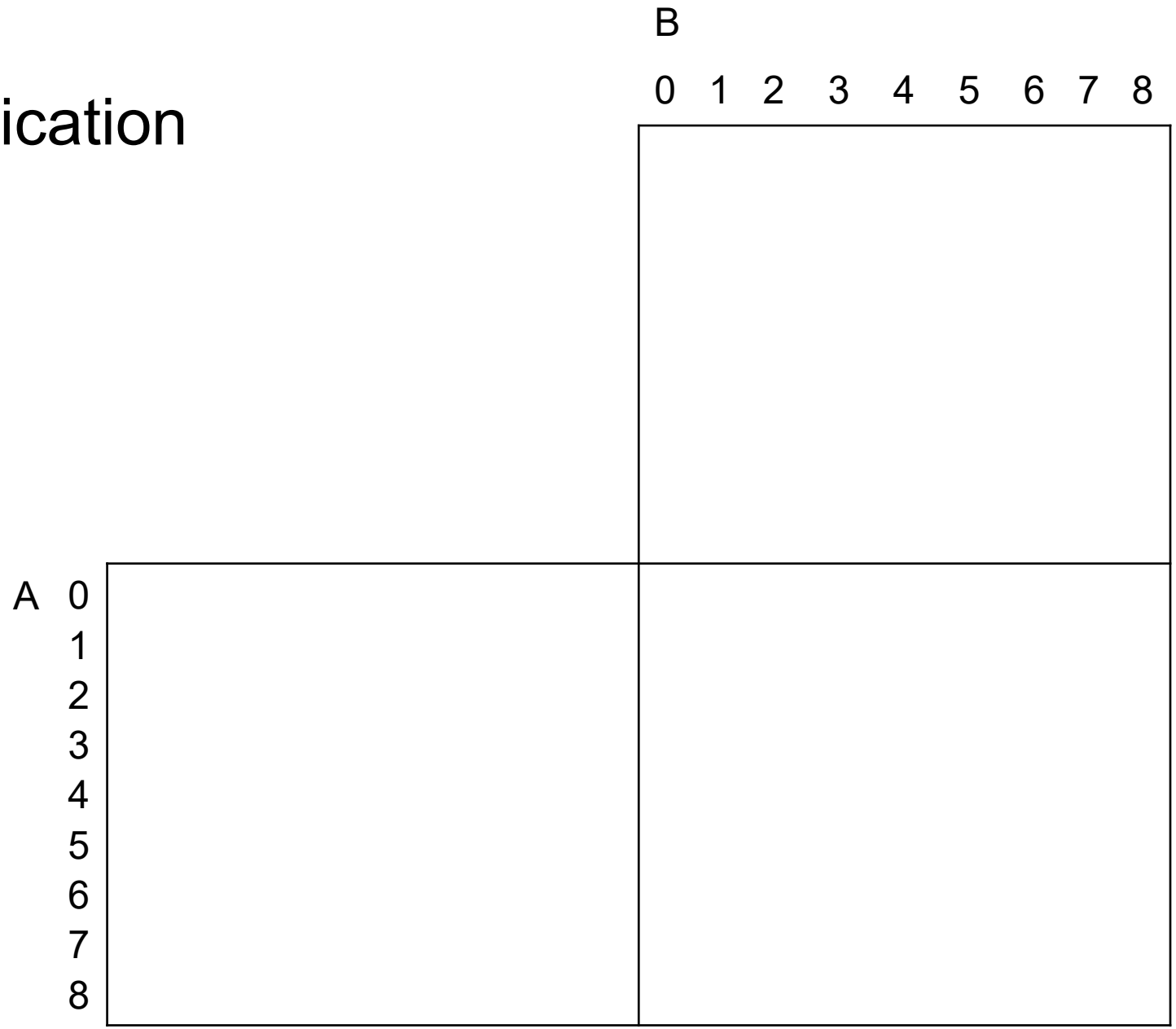
- A submission script
- An executable
- Input files needed by the executable
- Output files created by the executable
- Scheduling
  - FIFO
  - Job priority by processor count

Not again….

MATRIX MULTIPLICATION

A problem

- Matrix multiplication



Shared memory model

OpenMP:

```
#pragma omp parallel for
for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
                for(int k=0; k<dimension; k++){
                        matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                }
        }
}
```

Consider:

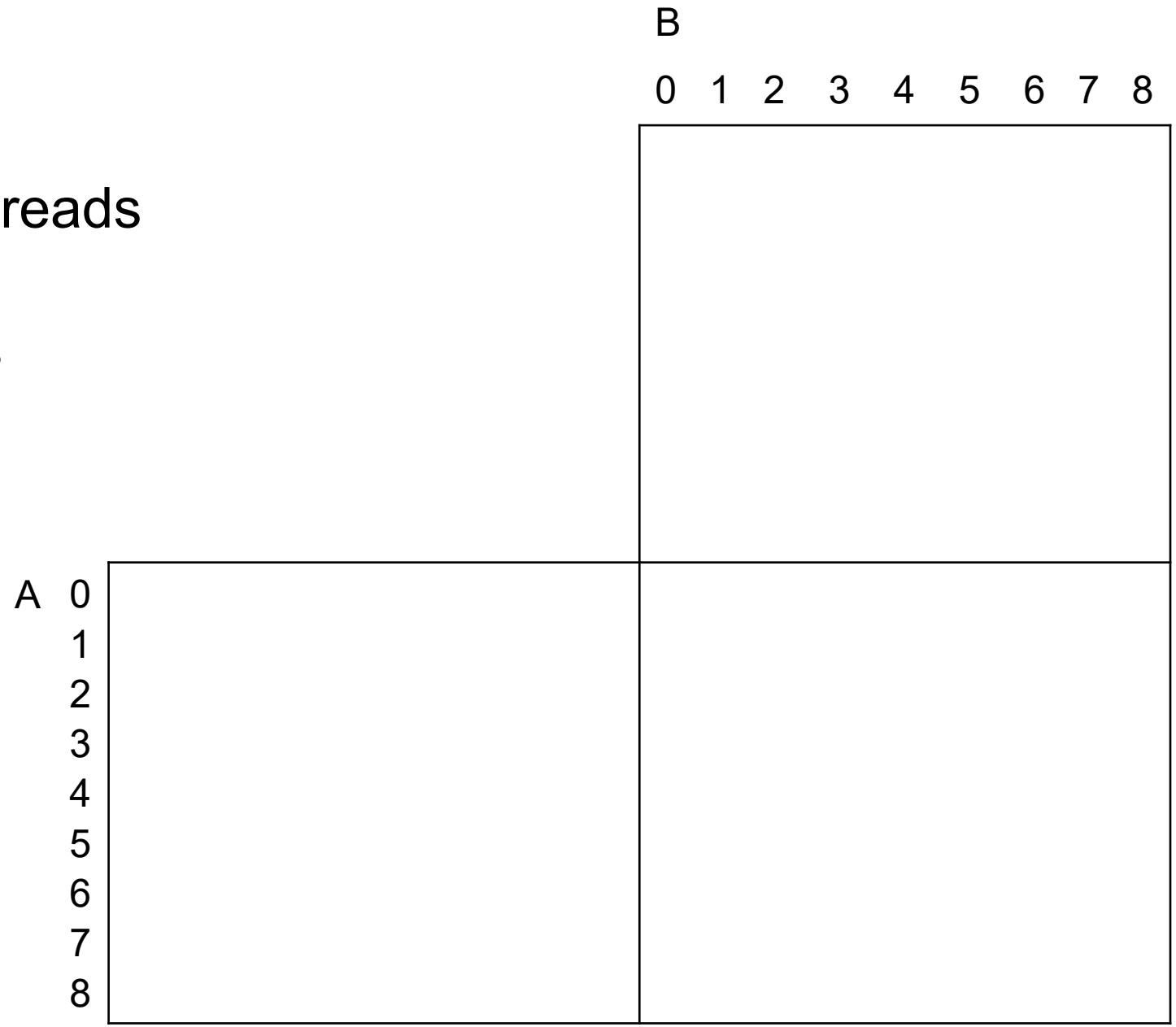
- Task granularity
- Memory – heap vs stack
- Cache?

```
#pragma omp parallel for
for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
                flatA[i * dimension + j] = matrixA[i][j];
                flatB[j * dimension + i] = matrixB[i][j];
        }
}


for(i=0; i<dimension; i++){
        iOff = i * dimension;
        for(j=0; j<dimension; j++){
                jOff = j * dimension;
                tot = 0;
                for(k=0; k<dimension; k++){
                        tot += flatA[iOff + k] * flatB[jOff + k];
                }
                matrixC[i][j] = tot;
        }
}
```

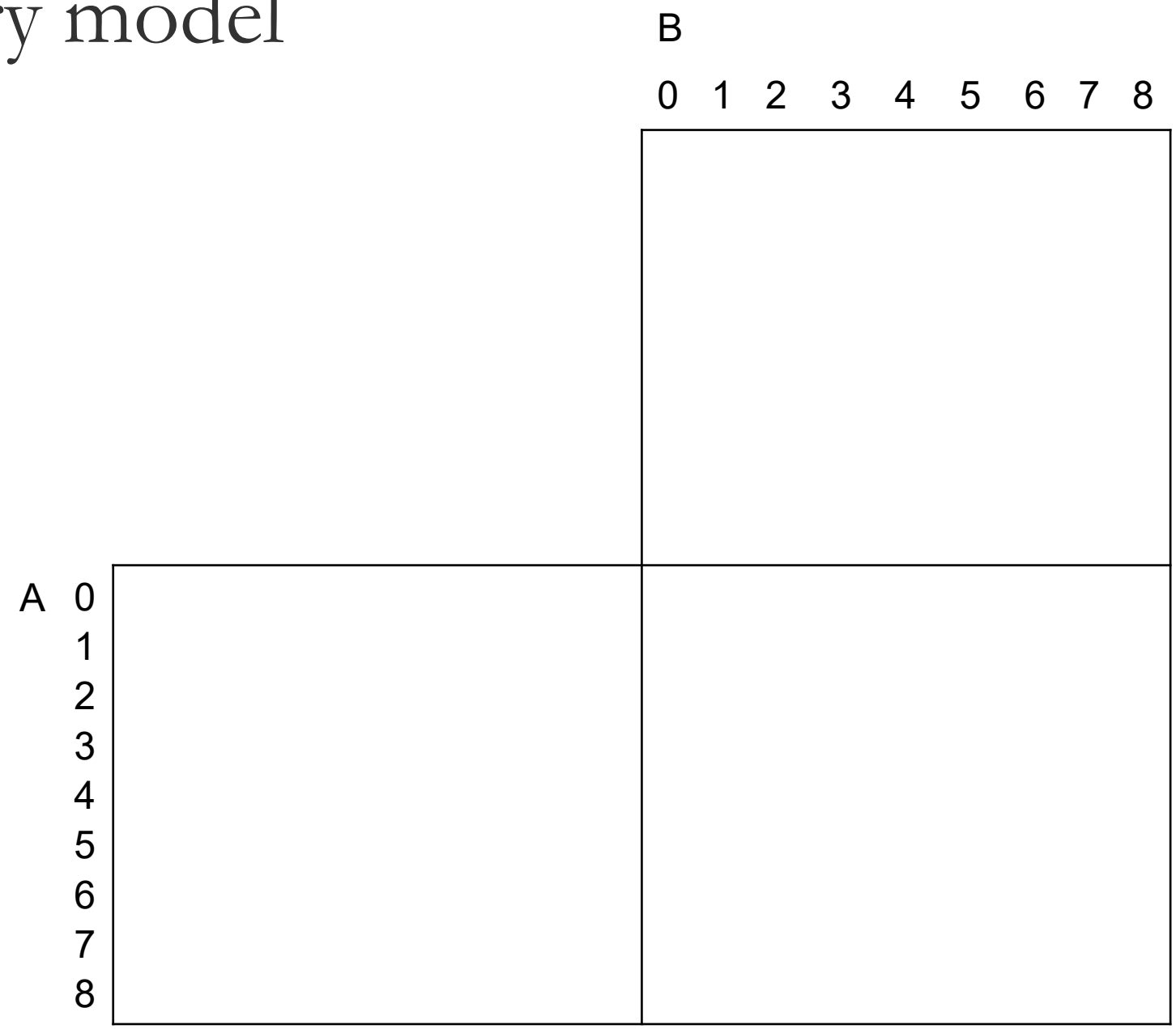
CUDA

- Consider
  - Number of threads
  - Memory
  - Control flow?

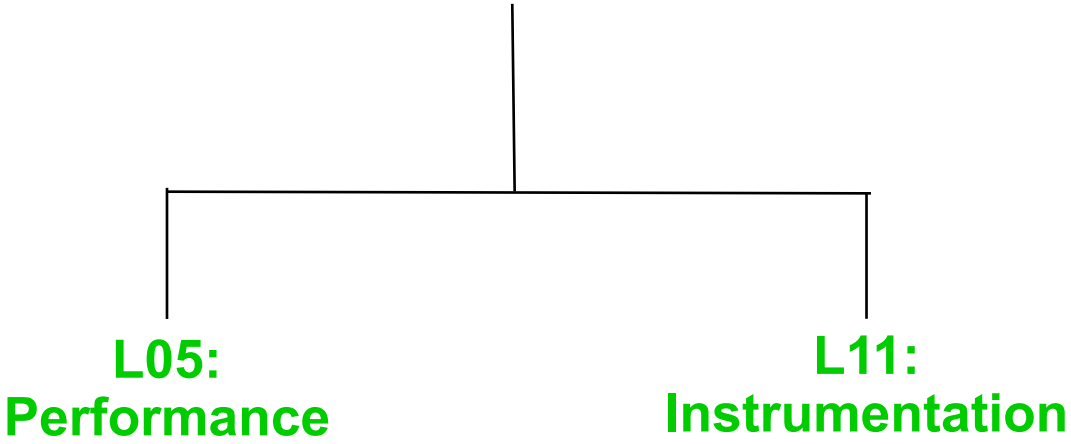


Distributed memory model

- MPI
- Consider:
  - Communication
  - Programming pattern
  - Data distribution
  - Interconnect?



PARALLEL PERFORMANCE



Performance

- **Goals and Factors**
- **Execution Time**
  - Sequential
  - Parallel
- **Speedup and Efficiency**
- **Scalability**
  - Fixed Problem Size – Amdahl's Law (1967)
  - Fixed Time – Gustafson's Law (1987)
- **Communication Time**

Parallel Program: **Cost**

- Cost of a parallel program with input size **n** executed on **p** processors:

$$C_p(n) = p \times T_p(n)$$

- $C_p(n)$ measures the total amount of work performed by all processors, i.e. **processor-runtime product**

- A parallel program is **cost-optimal** if it executes the **same** total number of operations as the **fastest** sequential program

Parallel Program: **Speedup**

- Measure the benefit of parallelism
  - A comparison between sequential and parallel execution time

$$S_p(n) = \frac{T_{best\_seq}(n)}{T_p(n)}$$

- Theoretically, $S_p(n) \leq p$ always holds

- In practice, $S_p(n) > p$ (super linear speedup) can occurs:
  - e.g. Better cache locality, early termination etc.

Parallel Program: **Efficiency**

- Actual degree of speedup performance achieved compared to the maximum

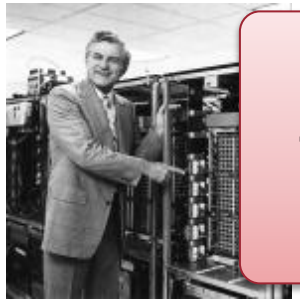
$$E_p(n) =$$

- We use T_* as a shorthand for T_{best_seq}
- Ideal speedup:

$$S_p(n) = p$$

$$E_p(n) = 1$$

Amdahl's Law (1967)



Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (***f***).

- ***f*** ($0 \leq f \leq 1$) is called the **sequential fraction**
- Also known as **fixed-workload performance**

- The most well-known law for discussing speedup performance
  - Applicable at all levels of parallelism

Gustafson's Law (1988)

- There are certain applications where the main constraint is execution time
  - Higher computing power is used to improve accuracy / better result

- If f is not a constant but decreases when problem size increases then

$$S_p(n) \leq p$$

Performance Instrumentation

- Look inside the different processing tasks, and analyze their timing, and energy usage!
- Our approach for analyzing timing
  - Analyse performance bottlenecks and tune codes to improve timing in normal/stressing scenarios

Understanding System's Performance

- Methodologies
- Instrumentation Tools
  - Observability Tools and Profiling
  - Debugging Tools
- Benchmarking

Methodologies

- Performance analysis in 60 seconds
- USE method
- CPU Profile Method
- Resource Analysis
- Others
  - Drill-down analysis
  - Off-CPU analysis
  - Static performance tuning
  - …

Profiling

- Use perf/perf_events
- Other tools:
  - Gprof
  - Vtune

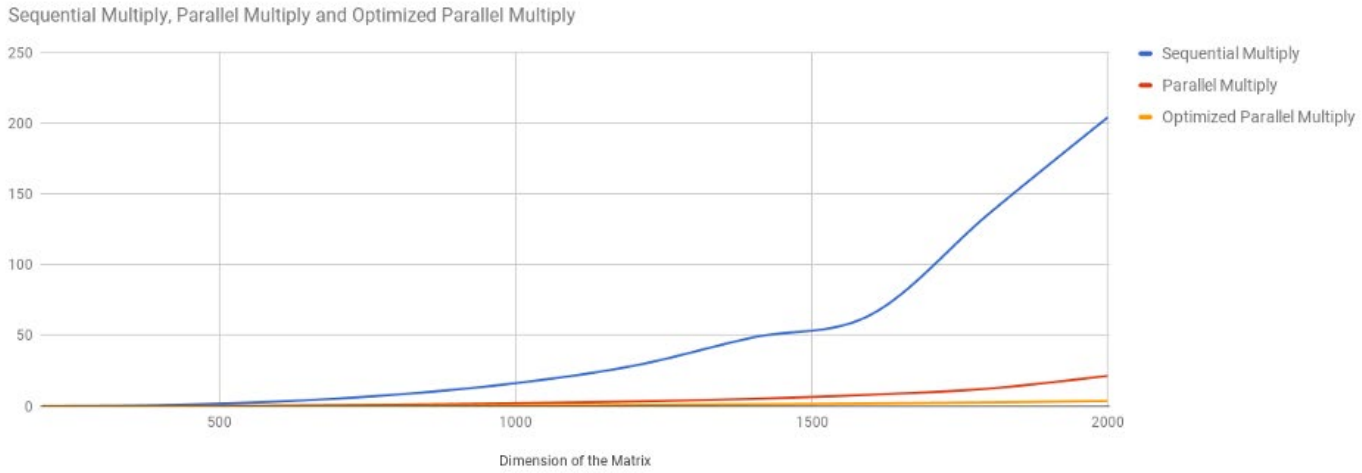
Debugging Tools

- Help in identifying bugs
  - Valgrind
    - Heavy-weight binary instrumentation: >4x overhead
    - Designed to shadow all program values: registers and memory
      - Shadowing requires serializing threads
    - Usually used for memcheck
  - Sanitizers
    - Compilation-based approach

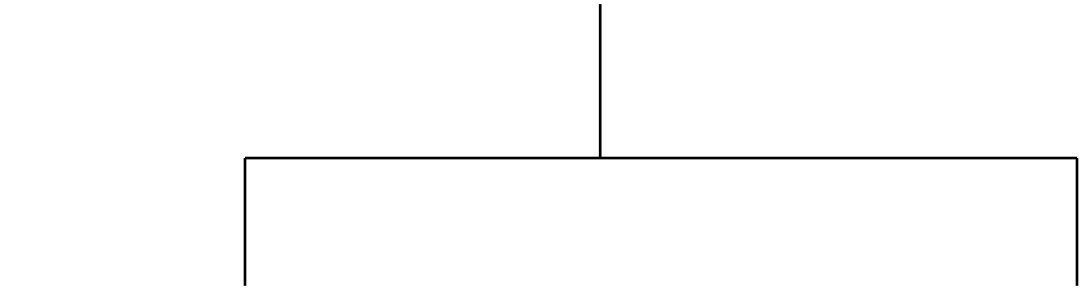
CS: Profiling on Summit

- Score-P - measurement infrastructure is a highly scalable
  - tool suite for profiling, event tracing, and online analysis of HPC applications
- Vampir – software performance visualizer focused on highly parallel applications
- NVIDIA Nsight for GPU profiling
  - Collect a timeline of the application

CS: Performance of Matrix Multiplication



NEW TRENDS

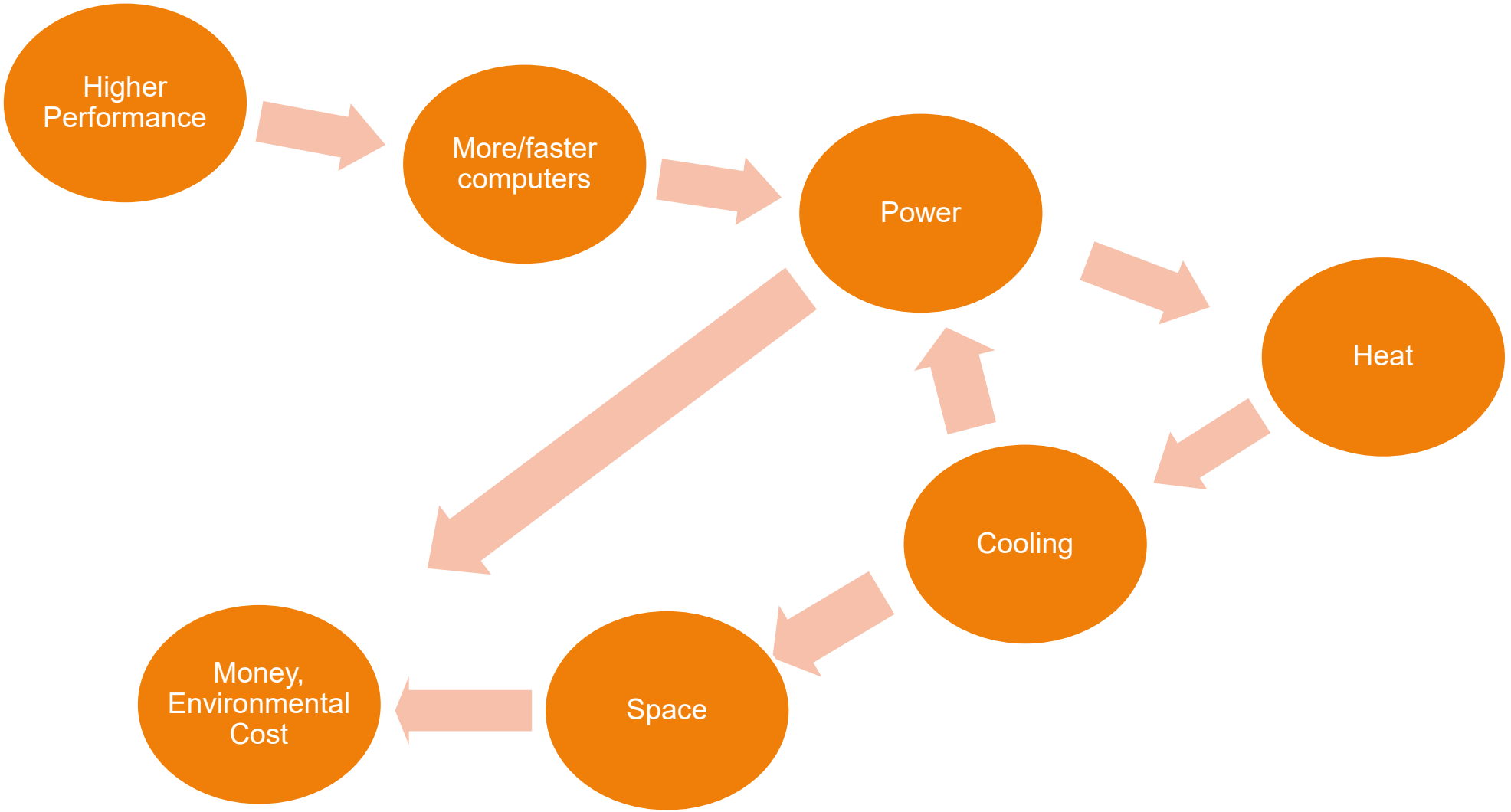


L12: Energy-efficient Computing

Energy Efficient Computing

- Energy-efficient Computing
  - Mobile computing
    - How ARM does it
    - Heterogeneous computing
  - Enterprise computing
    - Energy-efficient data center
      - How Google does it
    - Cloud computing
      - How Amazon does it

Costs of Computing



Data Centers - Factories of the digital age

- Efficiency
  - Storing, moving, processing, and analyzing data require energy
  - Double the energy consumption to keep the building from overheating
- Power Use Effectiveness (PUE): total amount of energy used in the center divided by the amount needed to run the processors
  - 2008: PUE in US was estimated to be 2.5
  - 2017: PUE in EU is estimated to be 1.8
  - Google: PUE of 1.1

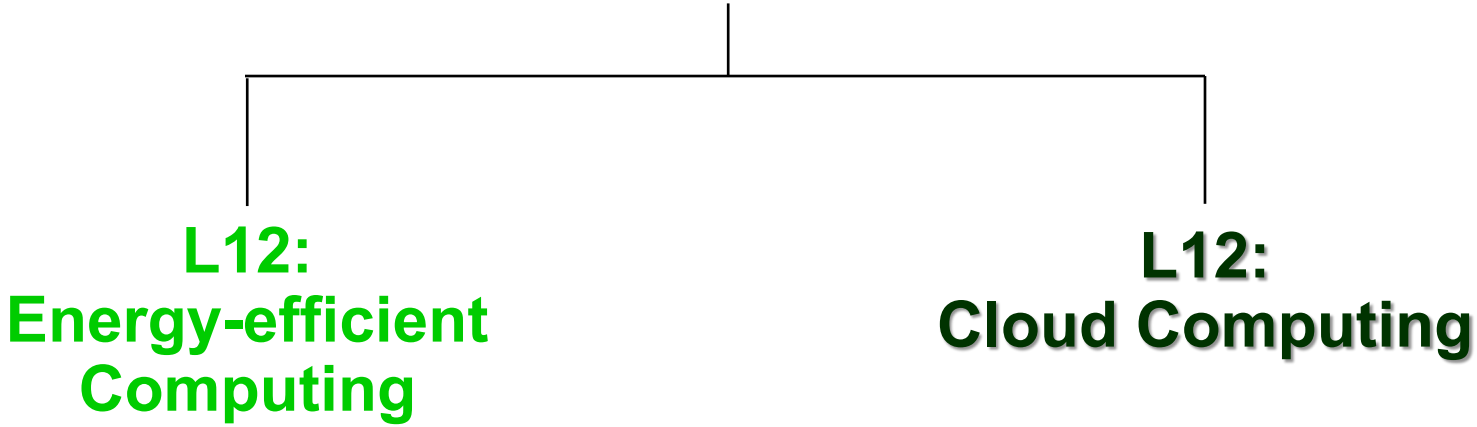
Energy Efficiency at Google

- Continuously measure efficiency
- Building custom, highly-efficient servers
- Extending the equipment lifecycle
- Controlling the temperature of the equipment
- Cooling with water instead of chillers

CS: Building Summit

https://vimeo.com/273729306

EVOLVING NEW MODELS

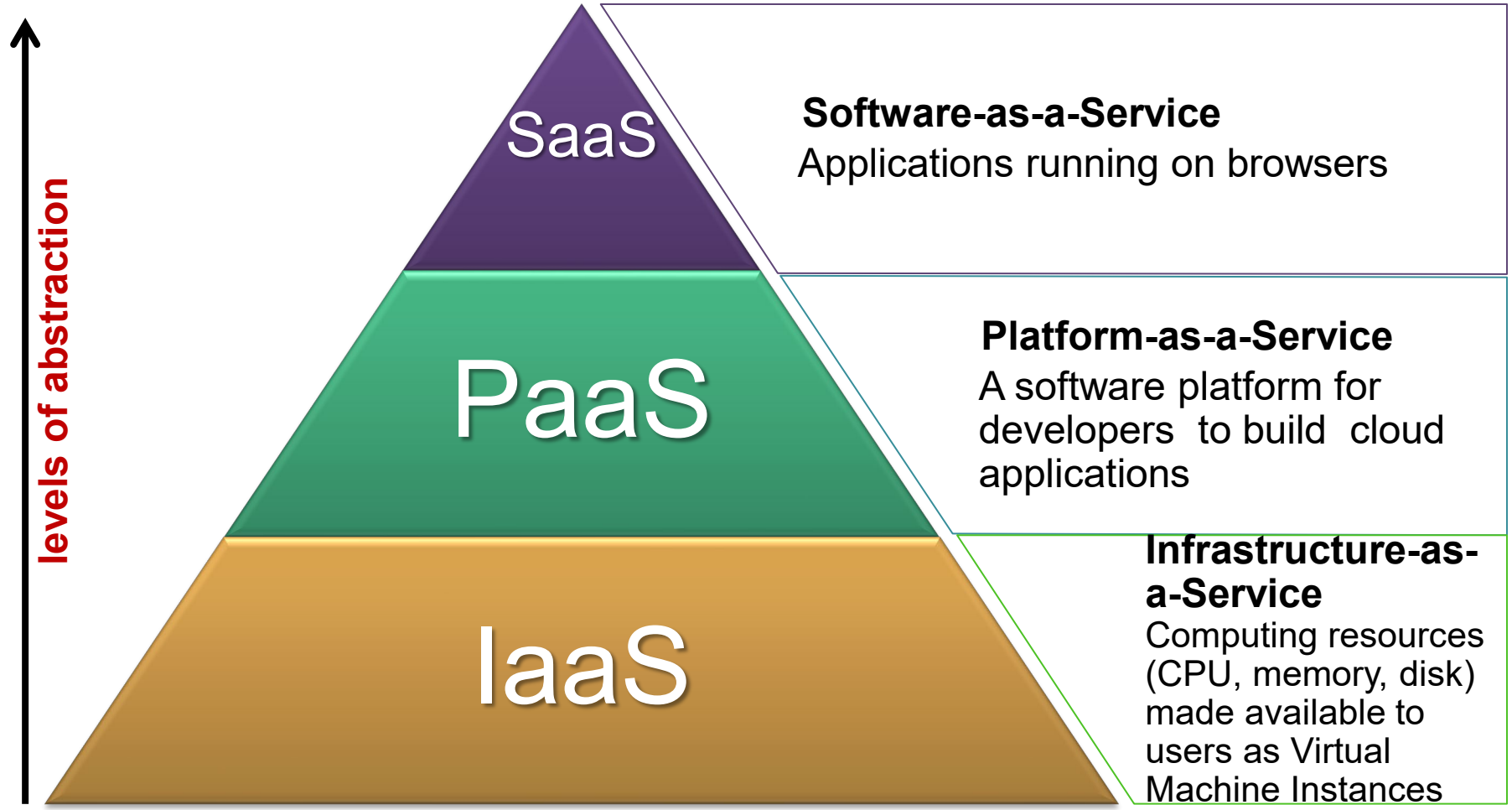


Cloud Computing: Virtualization

- Creation of a **virtual version** of something, such as an operating system, computing devices (server), storage devices or network devices

- Access resource without being concerned with **where** or **how** the resource is physically located or managed

Cloud Service (Delivery) Models



What can you read after this?

- CS3211 Parallel and Concurrent Programming
- CS4223 Parallel Computer Architecture
- CS4231 Parallel and Distributed Algorithms
- CS4345 General-purpose Computation on GPU
- CS5224 Cloud Computing
- …….

For you……

FINAL ASSESSMENT

Consultations before Exam

- Email me to book a timeslot
  - On most days
  - Don’t wait until the last few days
- Open book, Luminus file
  - Similar procedure with the midterm, but you need to record your screen
- Exam: 30 Nov, **9am**

Thank you!

- For being patient and understanding during these times
- For not losing your sense of humour

- Stay Healthy and Happy!