Introduction to Parallel & Distributed Computing

Programming Models & Methodologies

Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

Outline

- Parallel models
  - and the underlying architectures
- Design methodologies
  - increamental paralleization
  - Foster's methodology

All models are wrong, but some are useful.

-- George Box

Concept Clarification

- **Computational model**
  - an abstract model for exploring the power of computing
  - e.g., Turing machine, RAM, PRAM, BSP, LogP
- **Programming model**
  - programmer's view of a computing maching
- **Computer architecture**
  - hardware/software interface
- **Execution model**
  - how a program executes in runtime

random access machine

parallel RAM

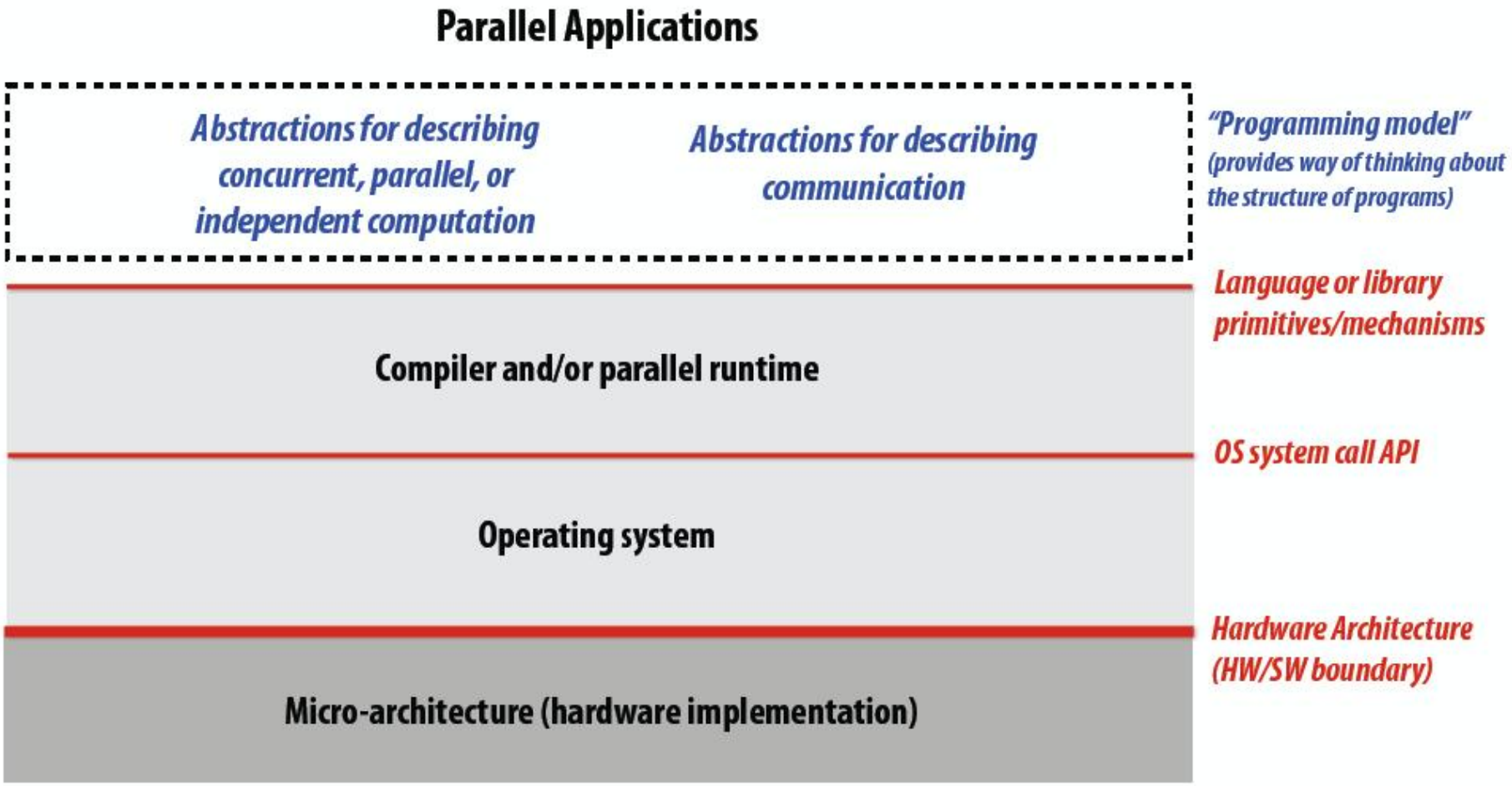
cuda

Parallel Models & Architectures

- **Three parallel programming models**
  - **That differ in communication abstractions presented to the programmer**
  - **Programming models influence how programmer think when they write programs**
- **Three machine architectures**
  - **Abstraction presented by the hardware to low-level software**
  - **Typically reflect hardware implementation's capabilites**
- **We'll focus on differences in communication and cooperation**

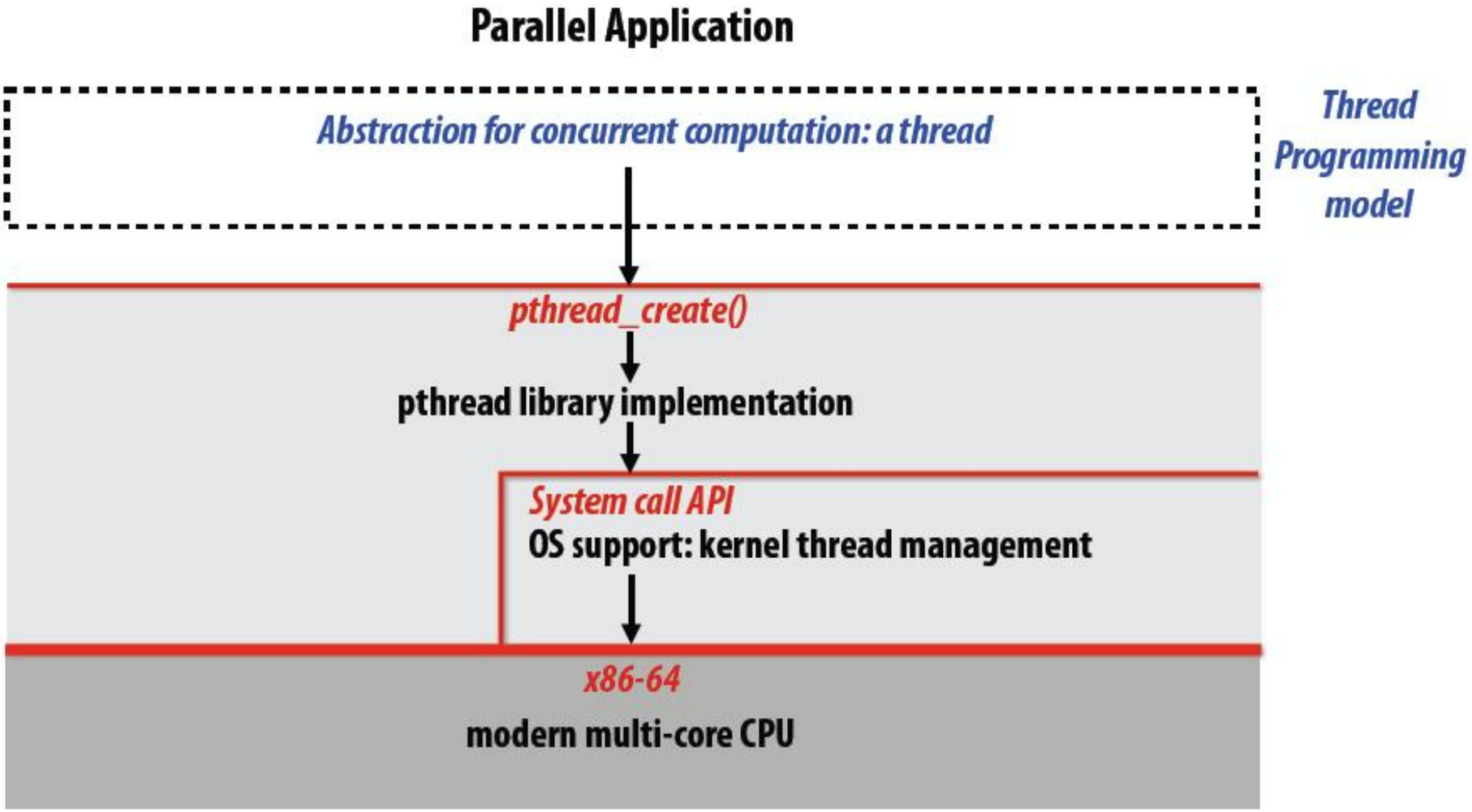
Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

System Layers: Interface, Implementation, Interface, …



Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Example: Expressing Parallelism with pthreads



Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Three Models of Communication

- **Shared address space**
- **Message passing**
- **Data parallel**

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Shared Address Space (SAS) Model

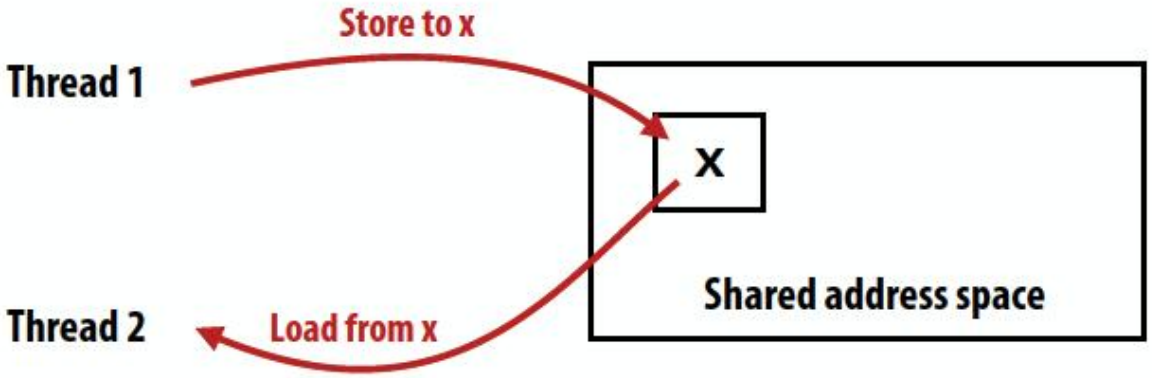
- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board
  - Any thread can read or write to shared variables

Thread 1:

```
int x = 0;
spawn_thread(foo, &x);
x = 1;
```

Thread 2:

```
void foo(int* x) {
  while (x == 0) {}
  print x;
}
```

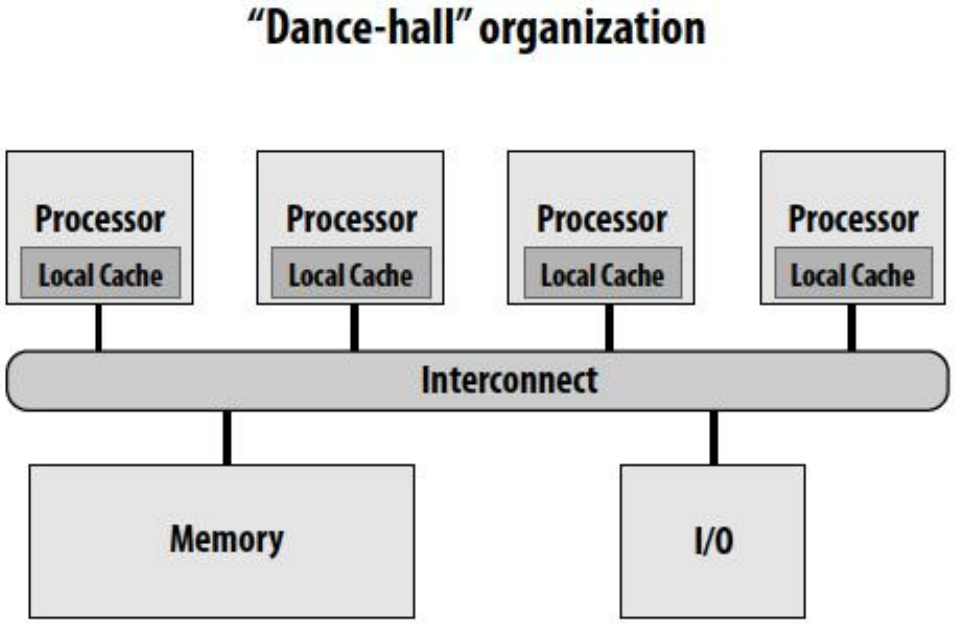


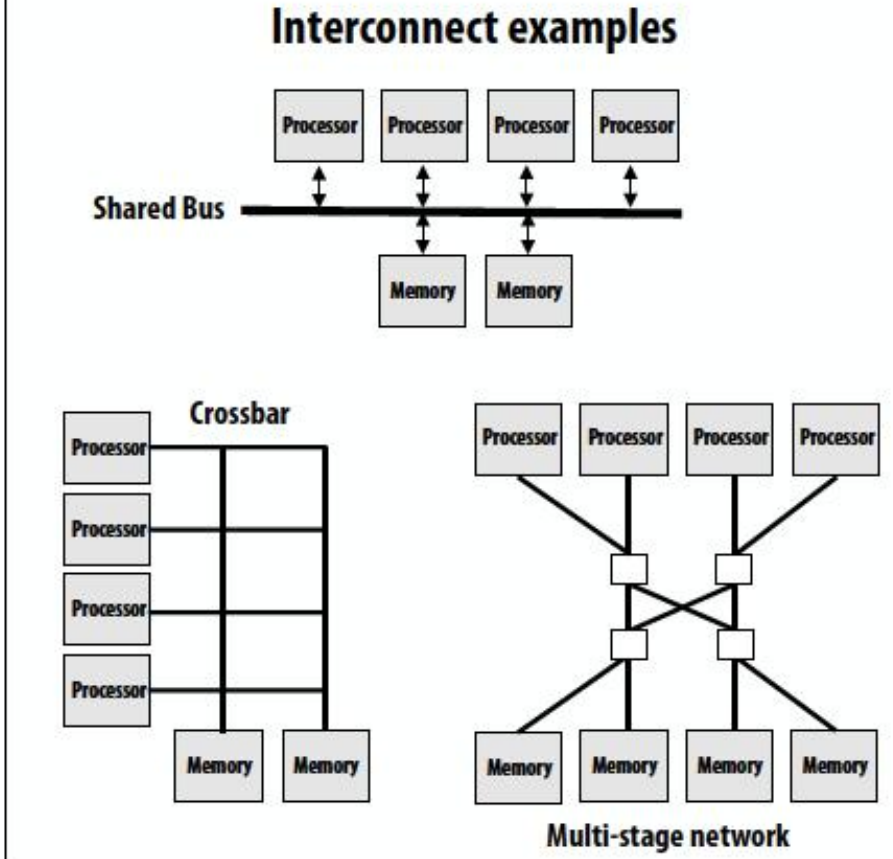
(Communication operations shown in red)

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

HW Implementation of an SAS

- **Key idea: any processor can directly reference any memory location**

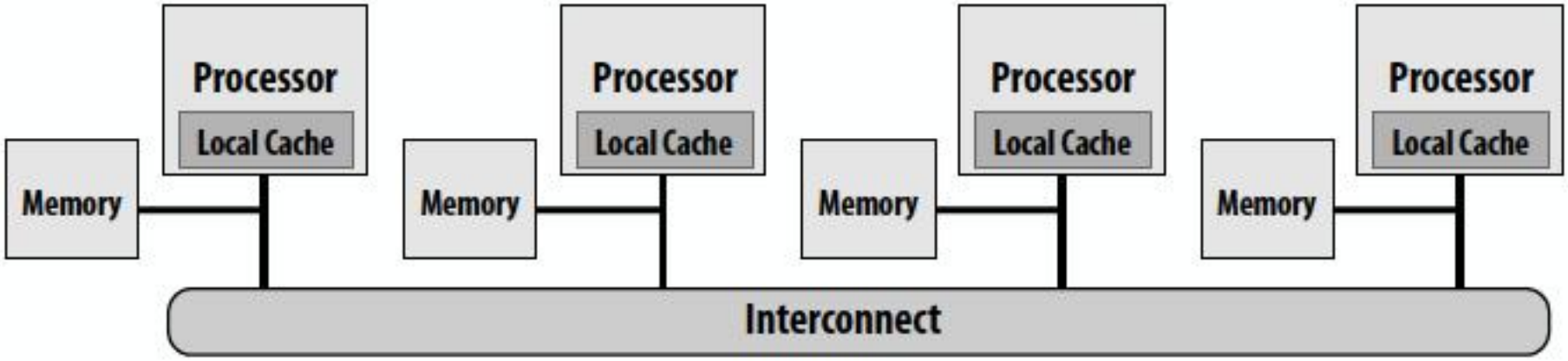




Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Non-uniform Memory Access (NUMA)

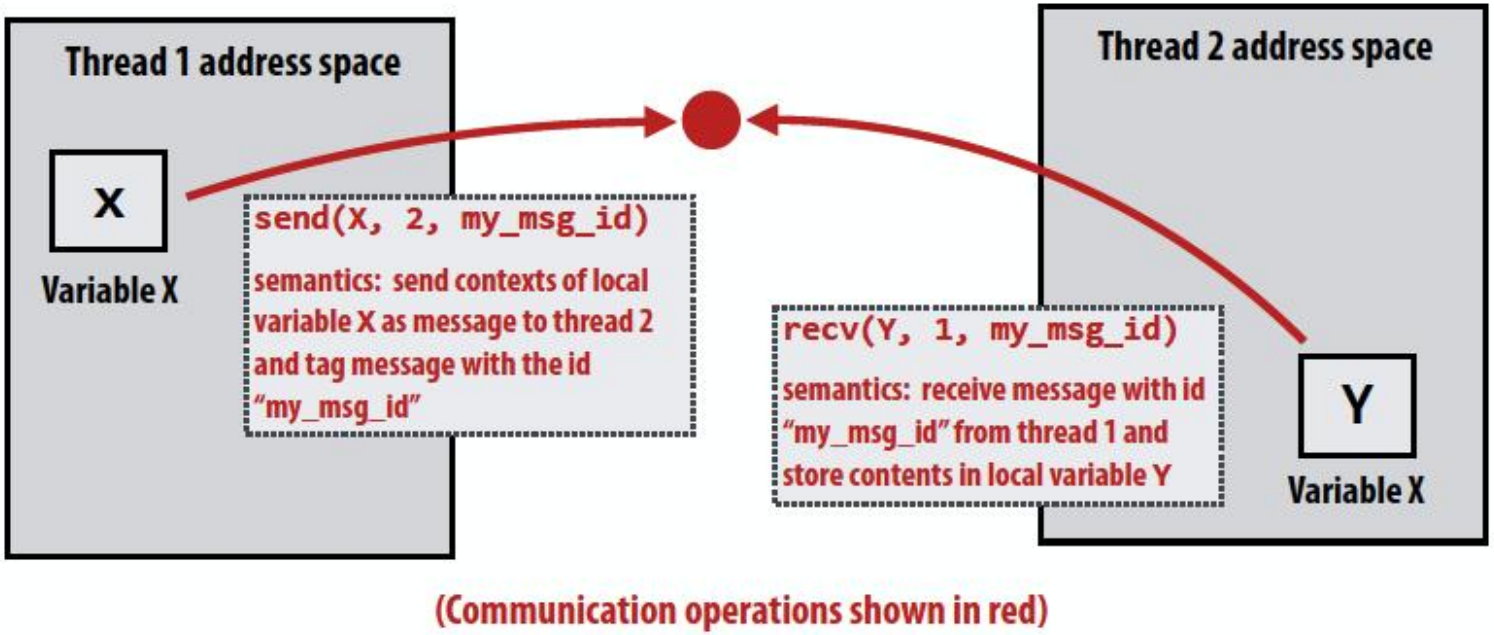
- Problem with preserving uniform access time in a system: scalability
- NUMA designs are more scalable
- Cost is increased programmer effort for performance tuning



Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Message Passing Model

- Threads operate within their private address spaces
- Threads communicate by sending/receiving messages



Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Message Passing

- Popular software library: MPI
- Hardware need not implement system-wide loads and stores to execute message passing programs
  - Need only be able to communicate messages

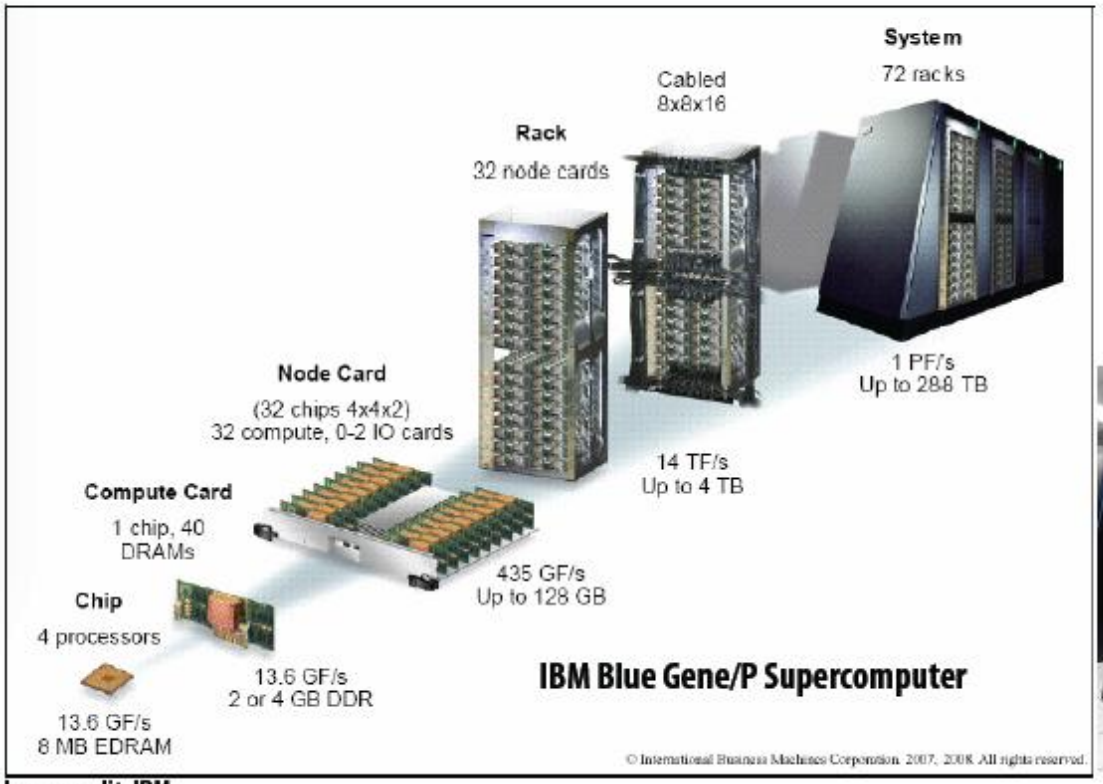
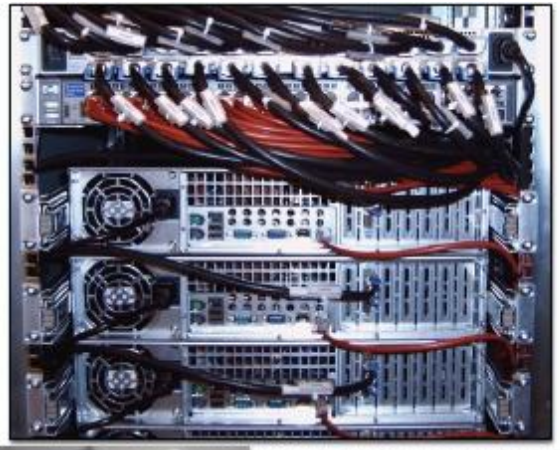


Image credit: IBM



Cluster of workstations
(Infiniband network)

The Correspondence between Programming Models and Machine Type is Fuzzy

- Common to implement message passing abstractions on machines that implement a shared address space in hardware
- Can implement shared address space abstraction on machine that do not support it in HW (via less efficient SW solution)
- Keep in mind
  - What is the programming model?
  - And What is the HW implementations?

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Programming Models impose Structure on Programs

- Share address space: very little structure
- Message passing: highly structured communication
- Data-parallel: very rigid computation structure
  - Programs perform same function on different data elements in a collection

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Data-Parallel Model

- **Historically: same operation on each element of an array**
  - **Cray supercomputers: vector processors**
    - **add(A, B, n): one instruction on vectors of length n**
- **Matlab is another good example: C = A+B**
  - **A, B, and C are vectors**
- **Today: often takes form of SPMD programming**
  - **map(function, collection), where**
  - **function is applied to each element independently**
  - **function maybe a complicate sequence of logic (e.g., a loop body)**
  - **Synchronization is implicit at the end of the map**
    - **map returns when function has been applied to all elements of collection**

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

Modern Practice: Mixed Programming Models

- **Very, very common practice**
  - Use shared address space programming within a multi-core node of a cluster
  - Use massage passing between nodes
- **In the future lectures, …**
  - CUDA/OpenCL use data-parallelism model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate
  - MapReduce use data parallelism to distribute communication among nodes

Source: K. Fatahalian and R. Bryant, CMU 15-418/618 Parallel Computer Architecture and Programming, Spring 2016

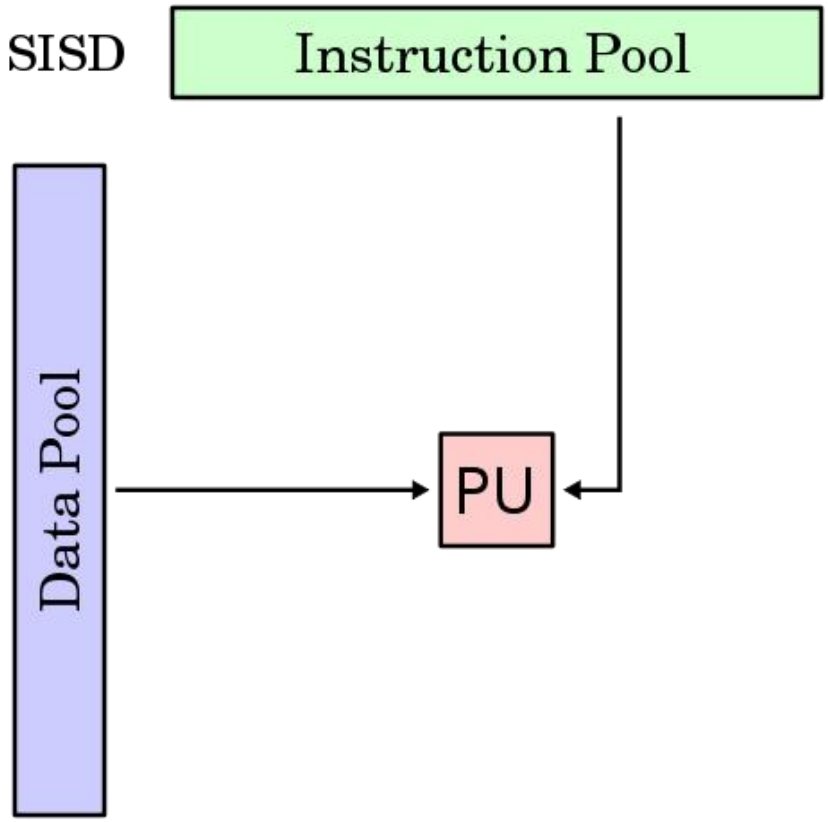
Flynn's Taxonomy

- **A classification of computer architectures based on the number of streams of instructions and data:**

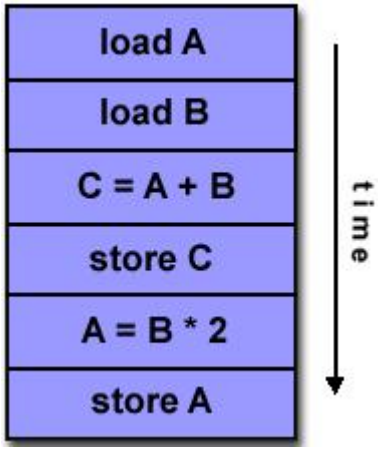
		Instruction stream	
		single	**multiple**
Data stream	**single**	SISD Single Instruction, Single Data	MISD Multiple Instructions, Single Data
	multiple	SIMD Single Instruction, Multiple Data	MIMD Multiple Instructions, Multiple Data

- **Program model converges with SPMD (single program multiple data)**

SISD Architecture



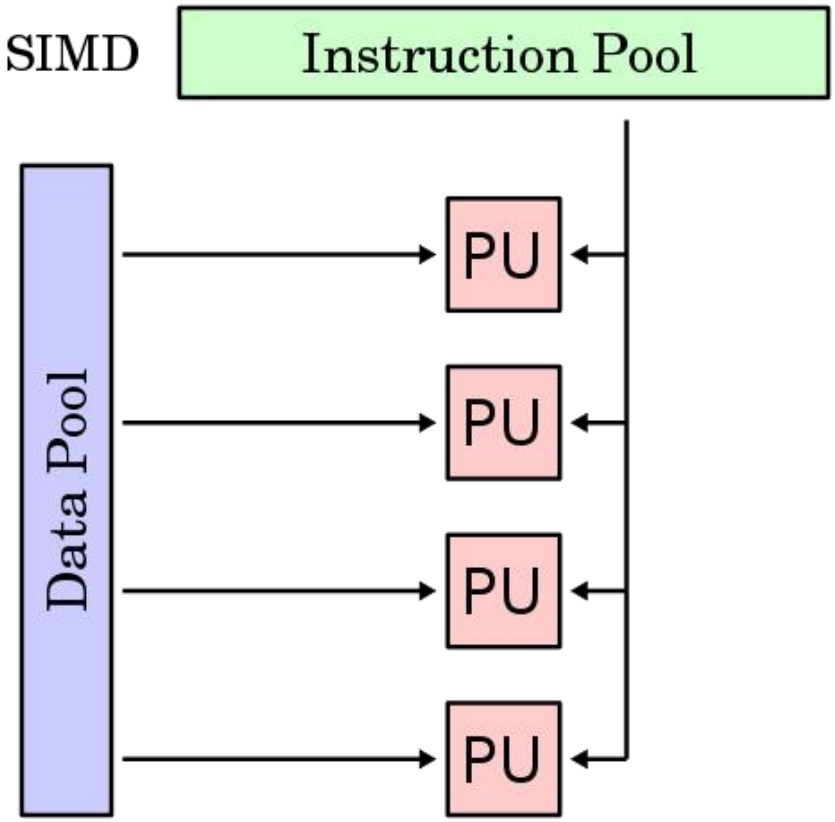
Example: single-core computers

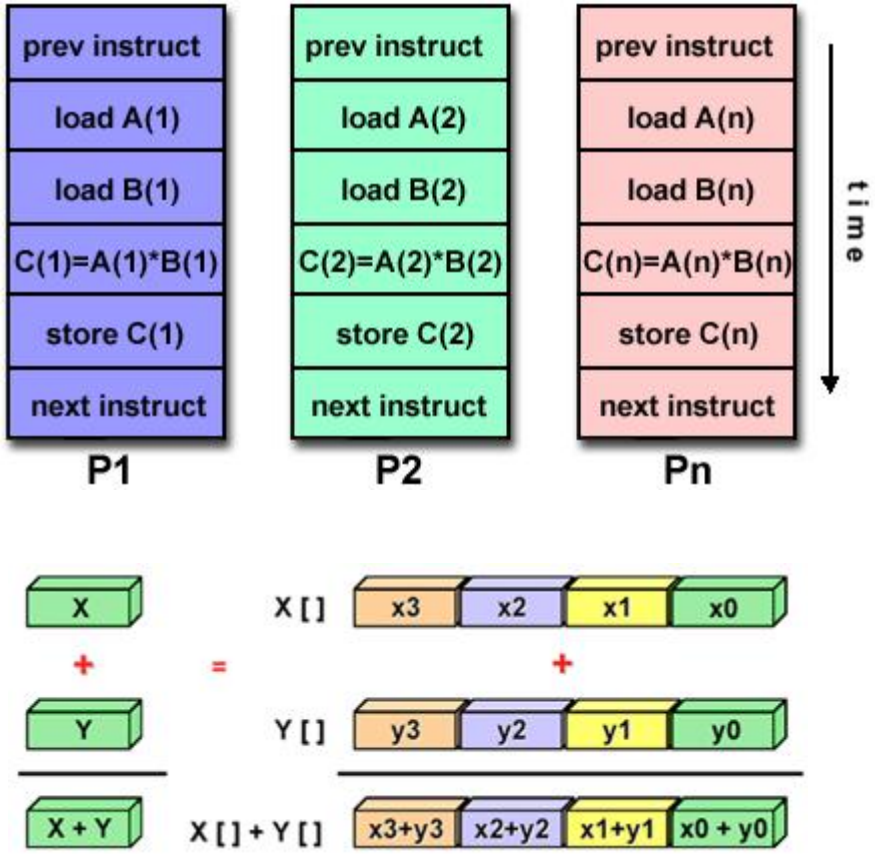


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

SIMD Architecture

Example: vector processors, GPUs

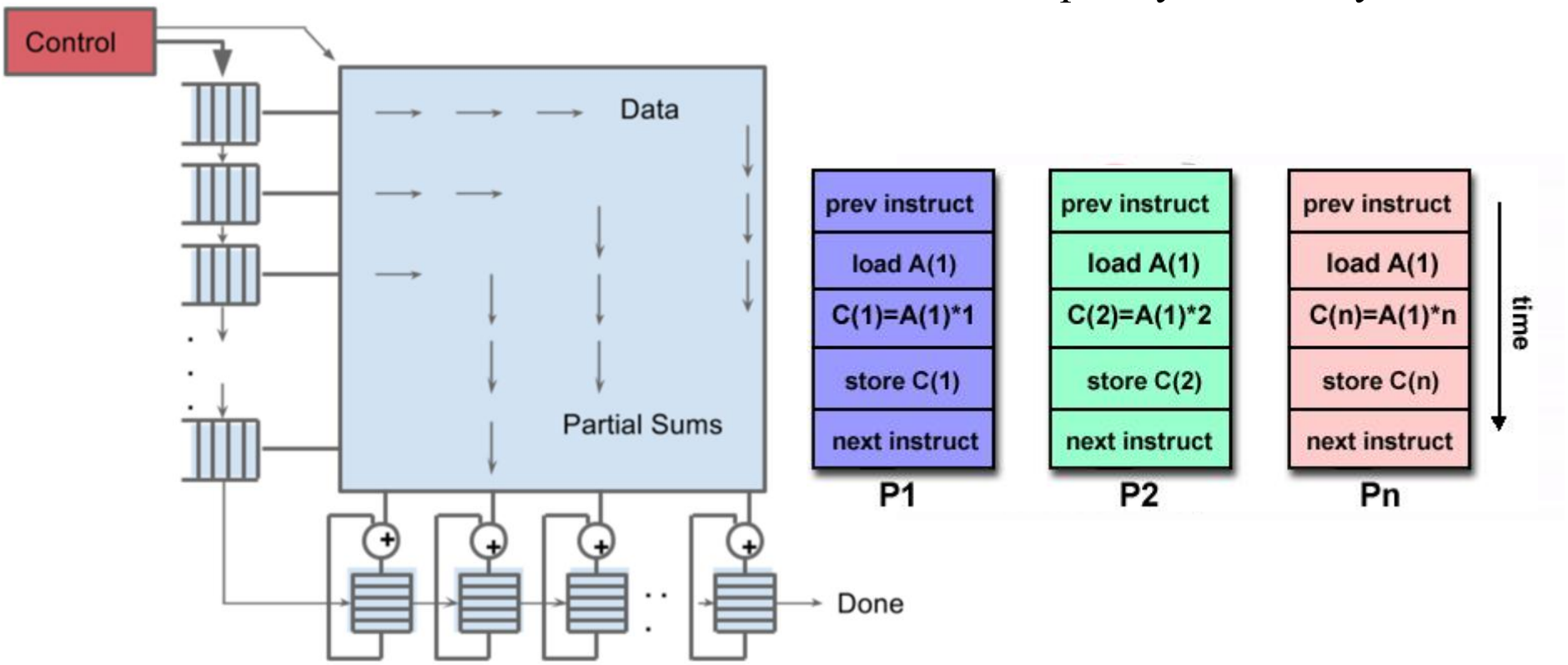




Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

MISD Architecture

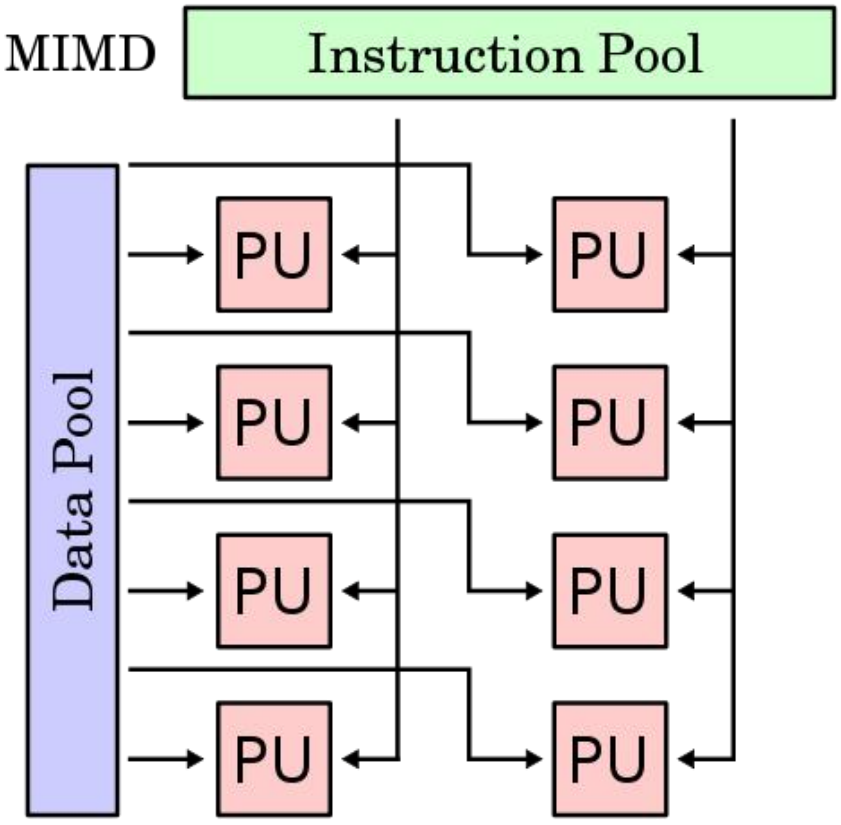
Example: systolic array

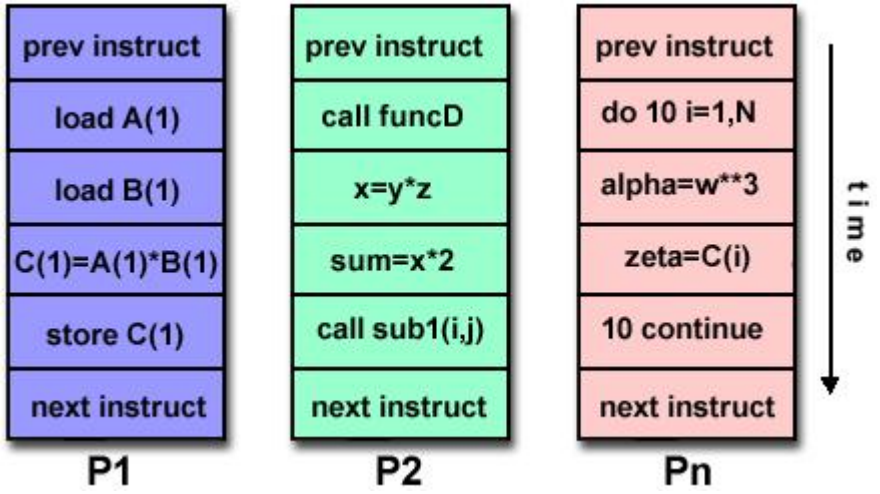


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

MIMD Architecture

Example: modern parallel systems





Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

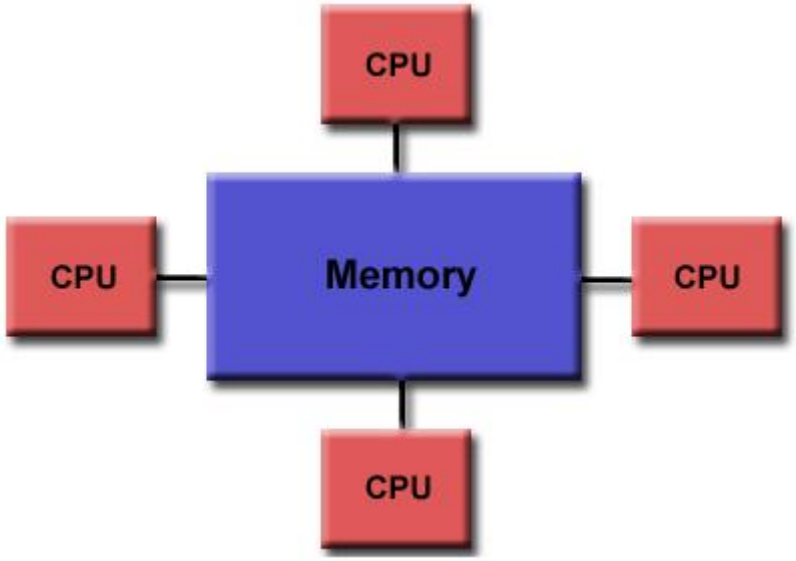
A Further Breakdown of MIMD

- **Shared-memory architecture**
  - **Symmetric multiprocessors (SMP), or uniform memory access (UMA)**
  - **Nonuniform memory access (NUMA) systems**
    - **Cache-coherent NUMA (ccNUMA)**
- **Distributed-memory architecture**
  - **Massively parallel processors (MPP)**
    - **Tightly coupled, specialized processors and network infrastructure**
  - **Clusters**
    - **Off-the-shelf computers connected by an off-the-shelf network**
- **Hybrid distributed-shared memory architecture**
- **Grids**
  - **Distributed, heterogeneous resources connected by LANs and/or WANs**

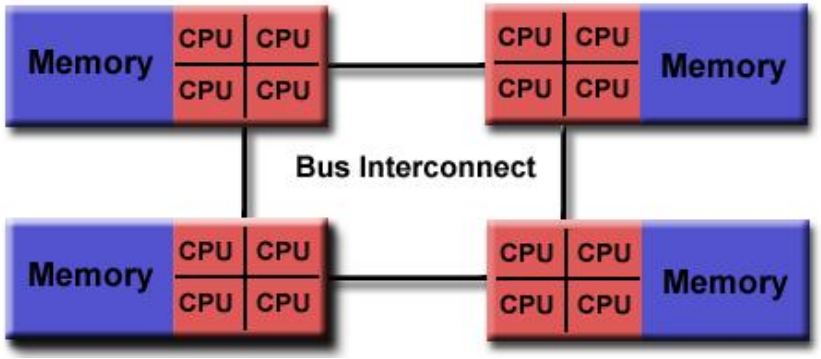
From T.G. Mattson, et al., "Patterns for Parallel Programming," Addison-Wesley Professional, 2004.

Shared-Memory Architecture

Symmetric Multiprocessors (SMP)

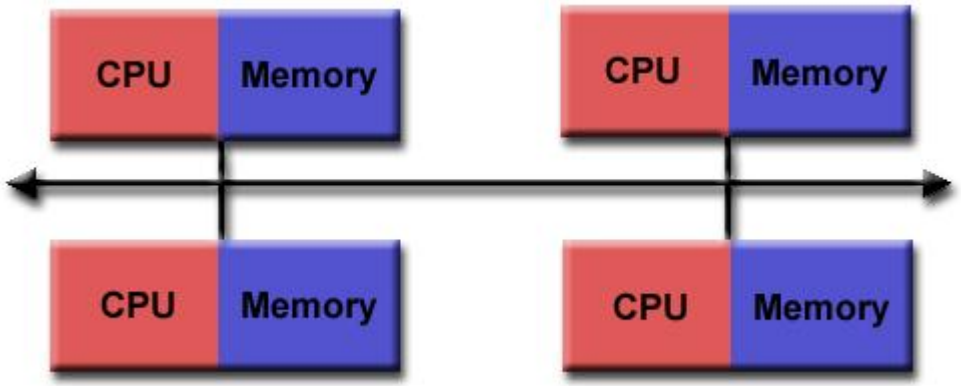


Nonuniform Memory Access (NUMA)



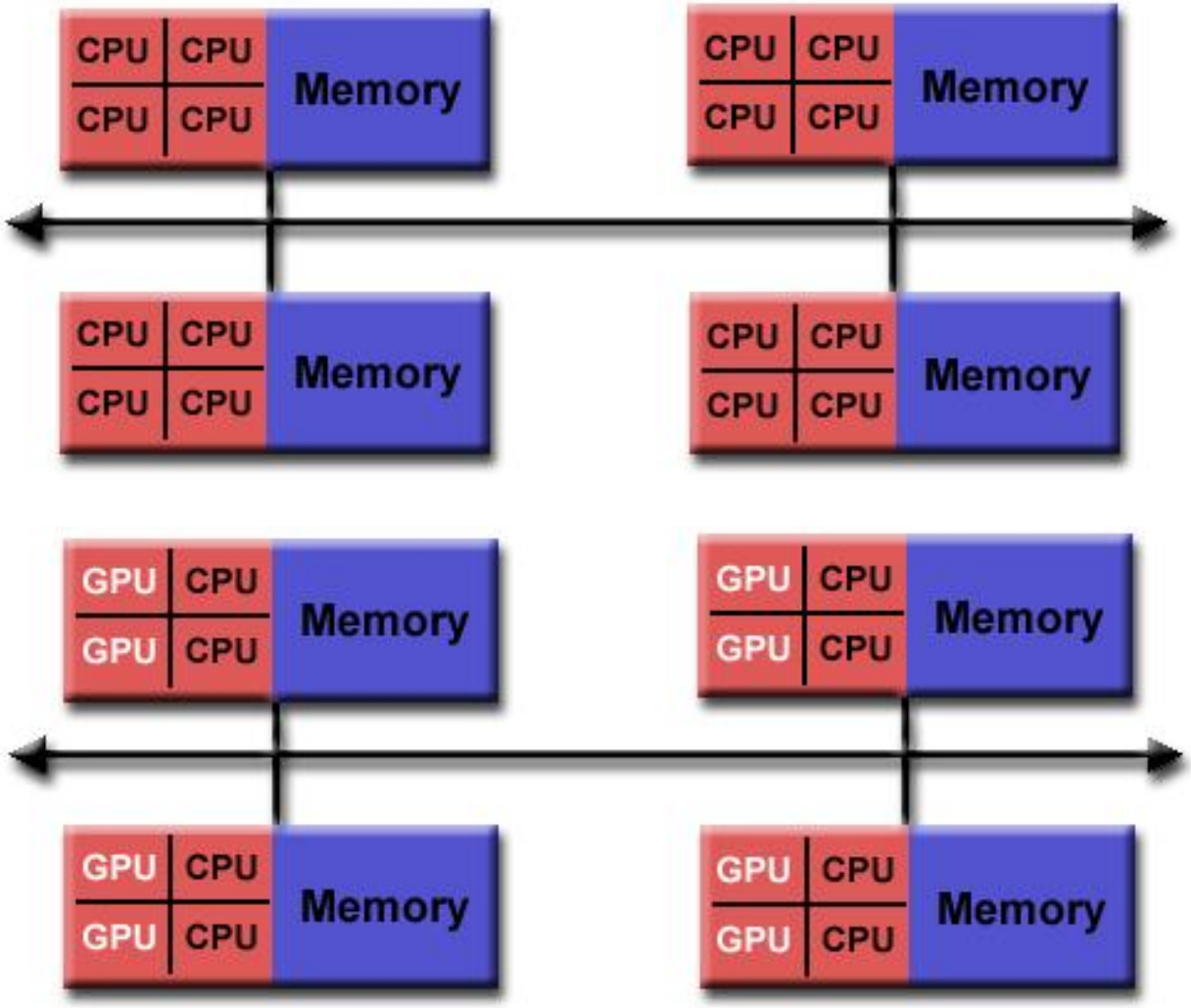
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Distributed-Memory Architecture



Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Hybrid Architecture

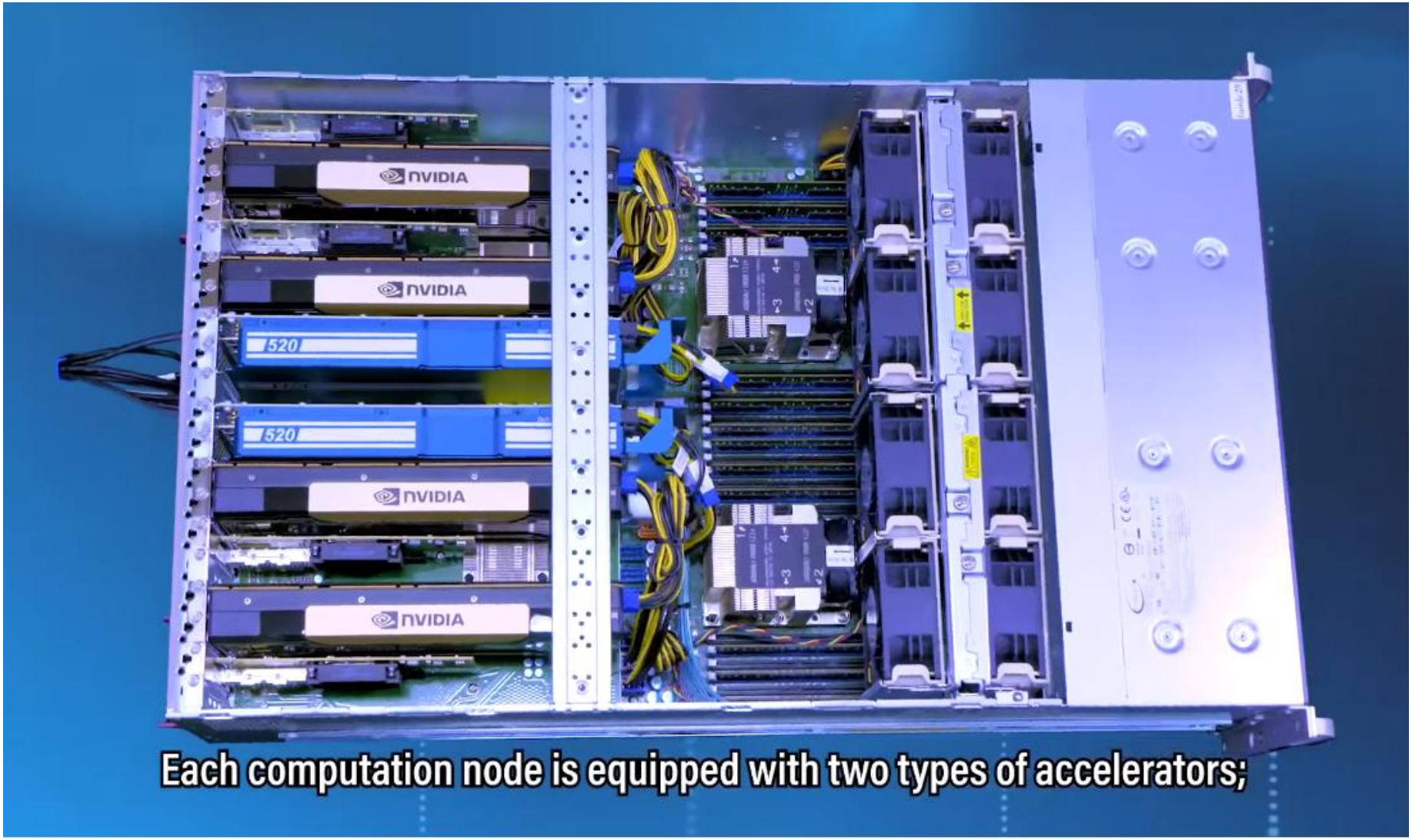


Hybrid & Homogeneous

Hybrid & Heterogeneous

Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Example: Tsukuba's Cygnus HPC



Outline

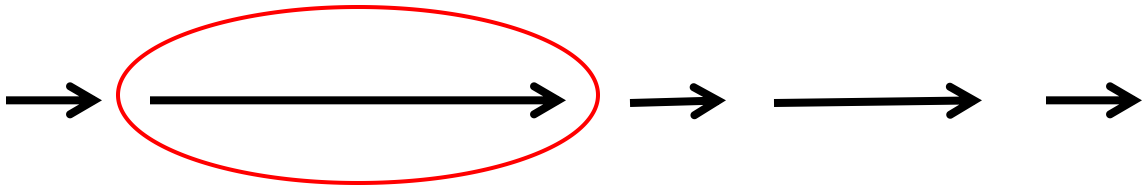
- Parallel models & architectures
- Design methodologies

Design Methodologies

1. **Incremental parallelization**
   - Study a sequential program (or code segment)
   - Look for bottlenecks & opportunities for parallelism
   - Try to keep all processors busy doing useful work
2. **Foster's Design Methodology**
   - Partitioning
   - Communication
   - Agglomeration
   - Mapping

Incremental parallelization

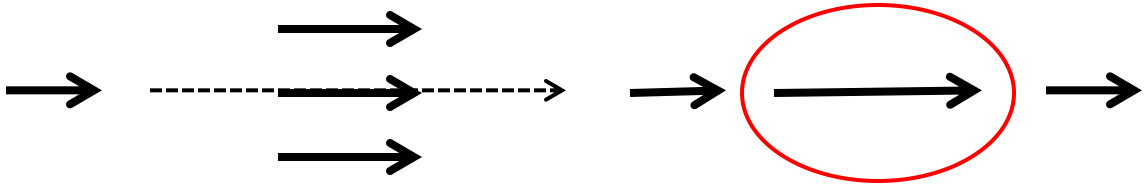
- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Incremental parallelization

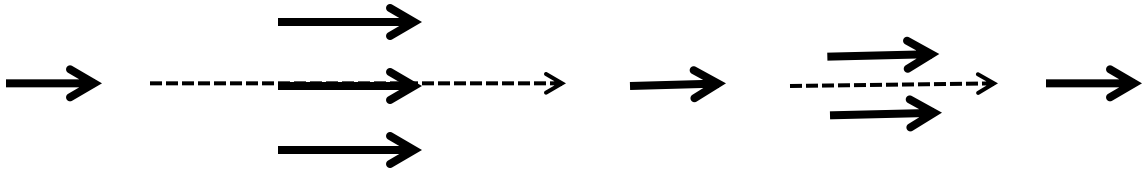
- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Incremental parallelization

- Study a sequential program (or code segment)
- Look for bottlenecks & opportunities for parallelism
- Try to keep all processors busy doing useful work

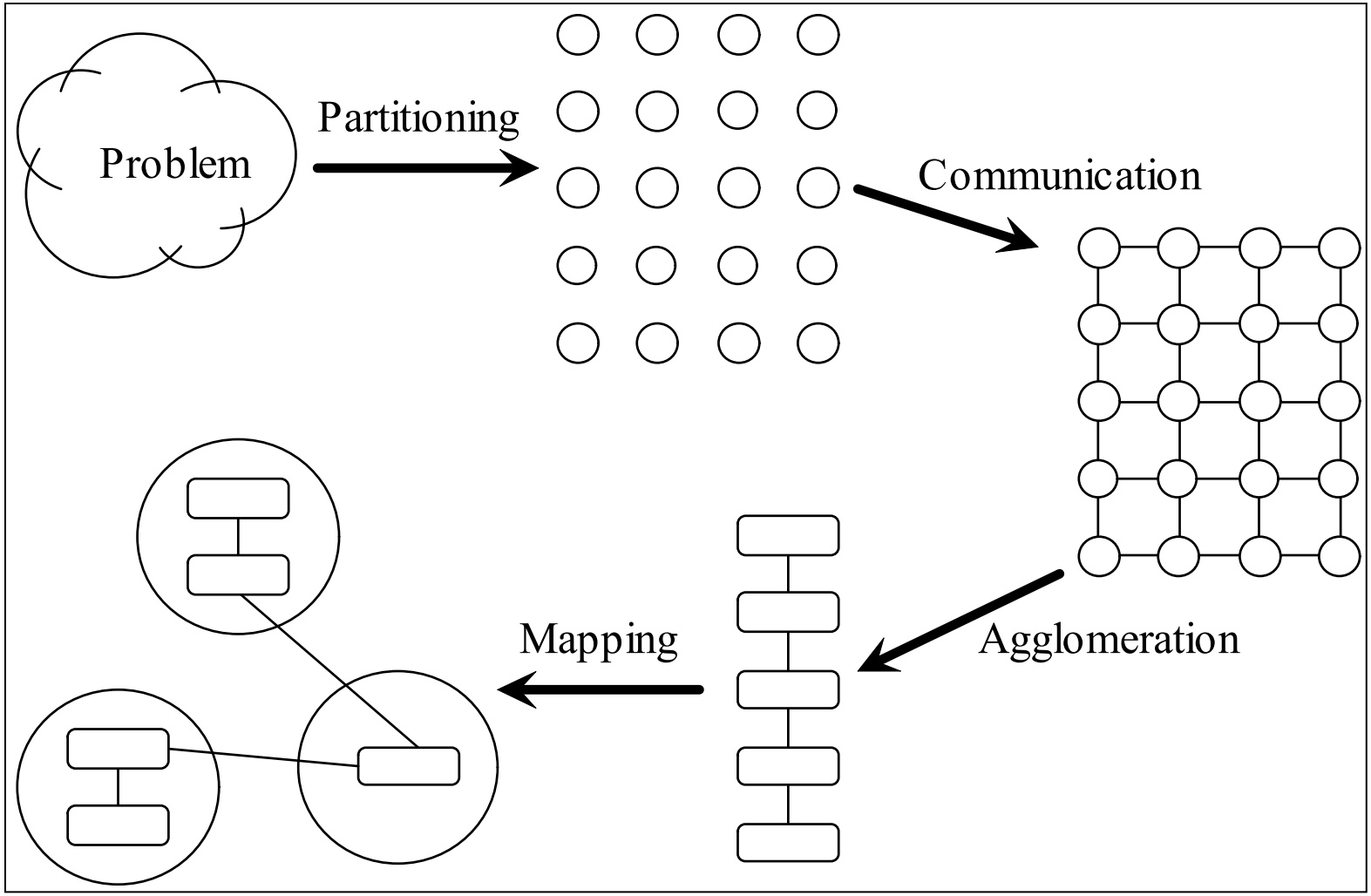


Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Foster's Design Methodology

- Partitioning
- Communication
- Agglomeration
- Mapping

Foster's Methodology



A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Partitioning

- Dividing computation and data into pieces
- Exploit data parallelism
  - (Data/domain partition/decomposition)
  - Divide data into pieces
  - Determine how to associate computations with the data
- Exploit task parallelism
  - (Task/functional partition/decomposition)
  - Divide computation into pieces
  - Determine how to associate data with the computations
- Exploit pipelining parallelism
  - (to optimize loops)

Data Partition

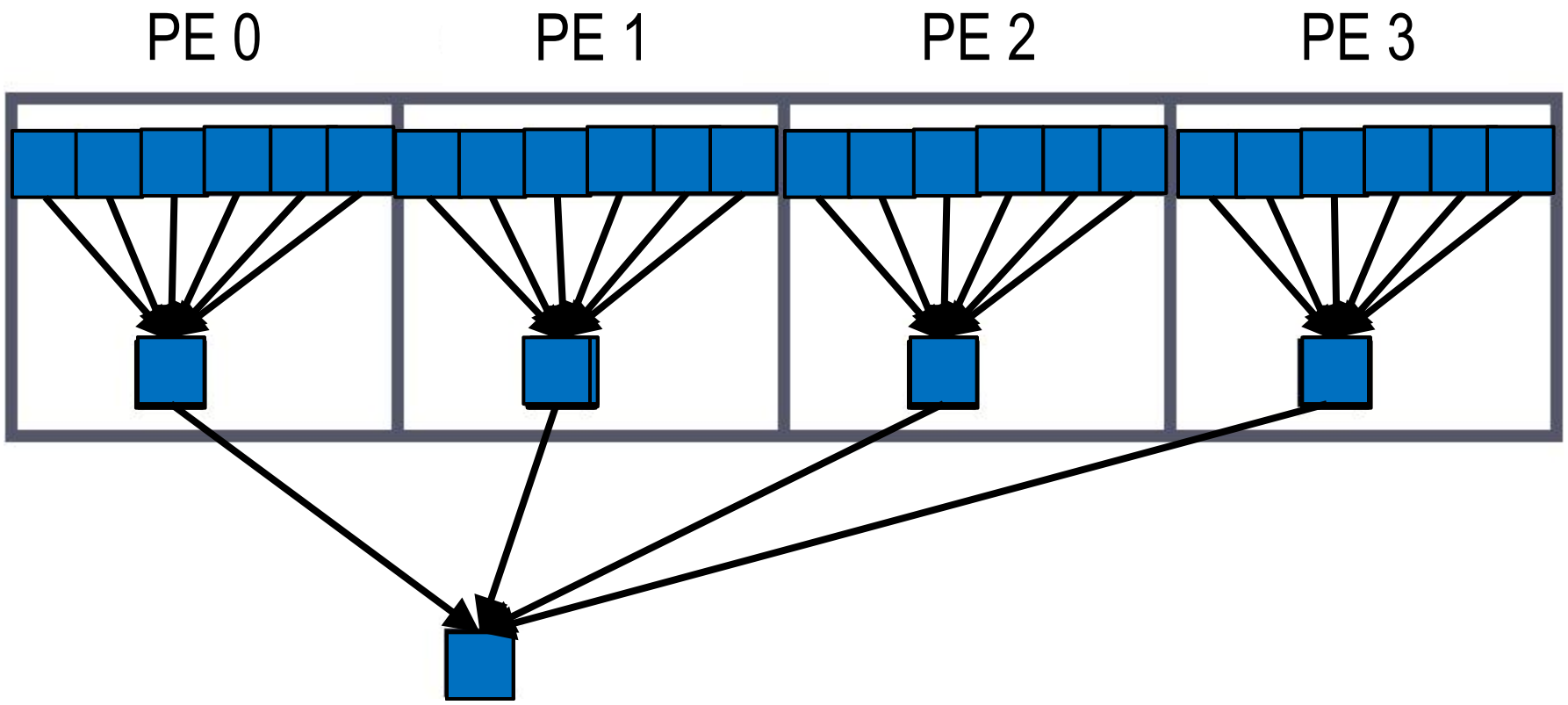
- First, decide how data elements should be divided among processors
- Second, decide which tasks each processor should be doing

- Example: find maximum element in a vector

Source: CS133 Spring 2010 at UCLA (Kaplan)

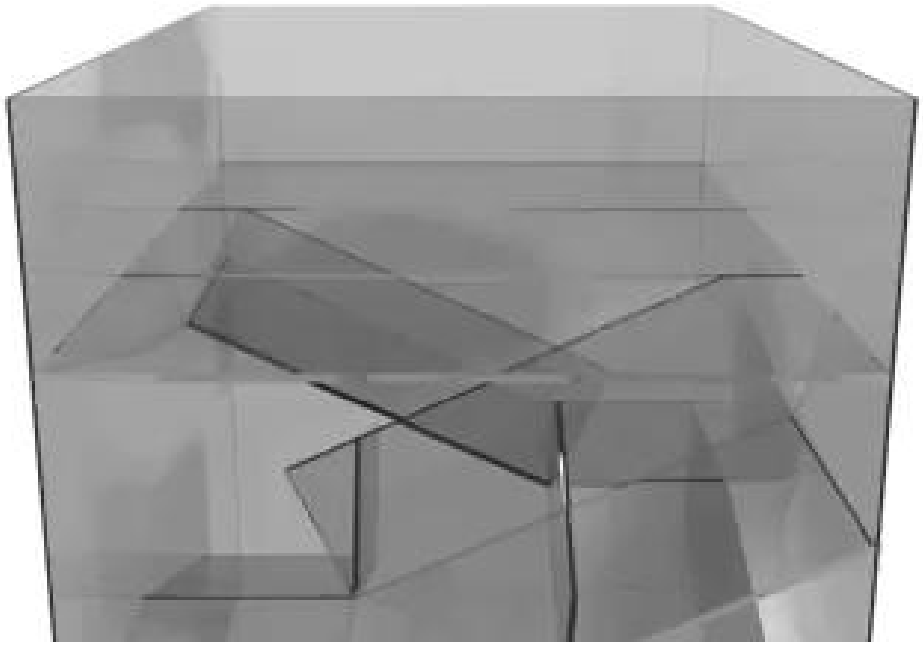
Data Partition: Example

Find the largest element of an array



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Data Partition: Another Example

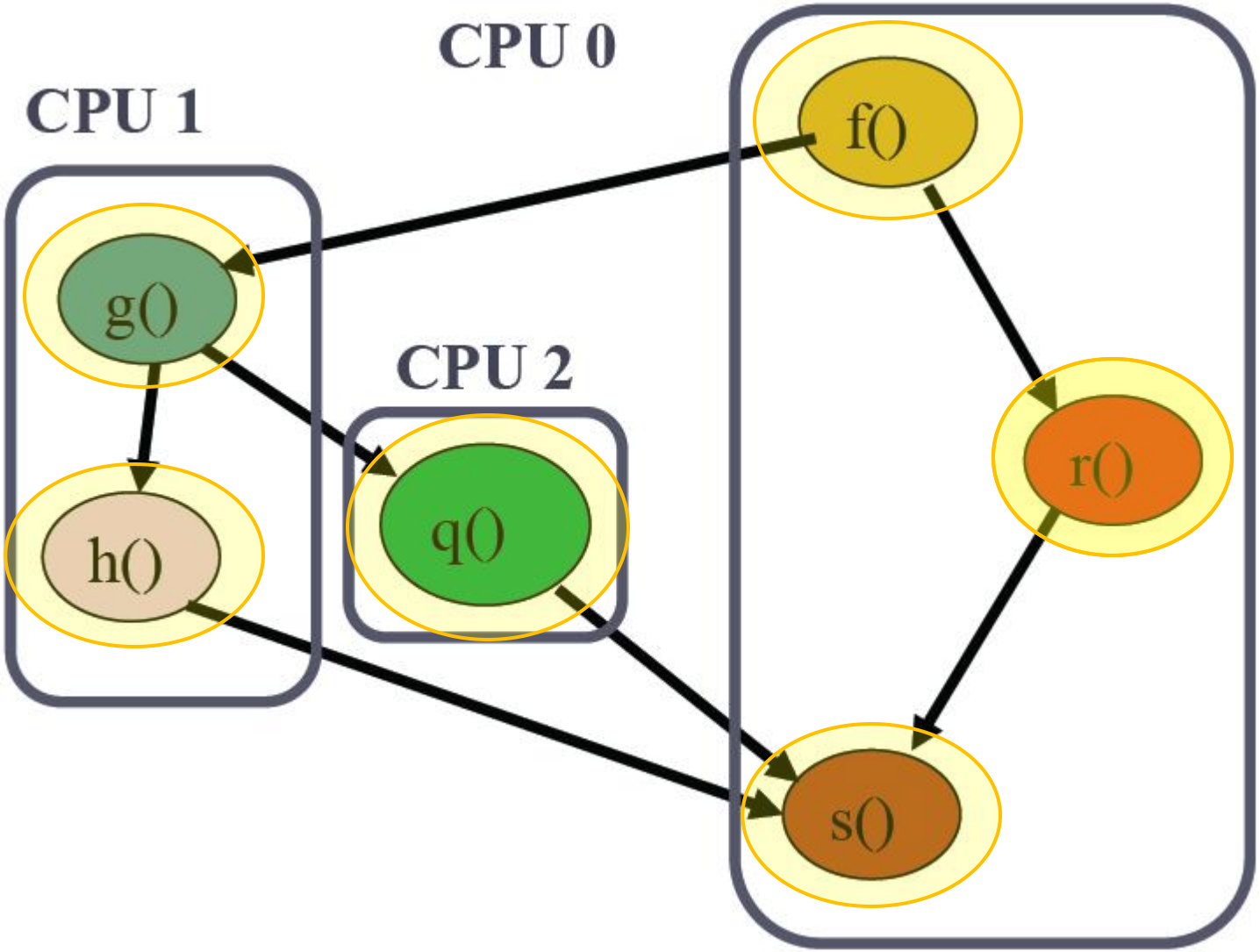


Task Partition

- First, divide tasks among processors
- Second, decide which data elements are going to be accessed (read and/or written) by which processor
- Example: event-handler for GUI

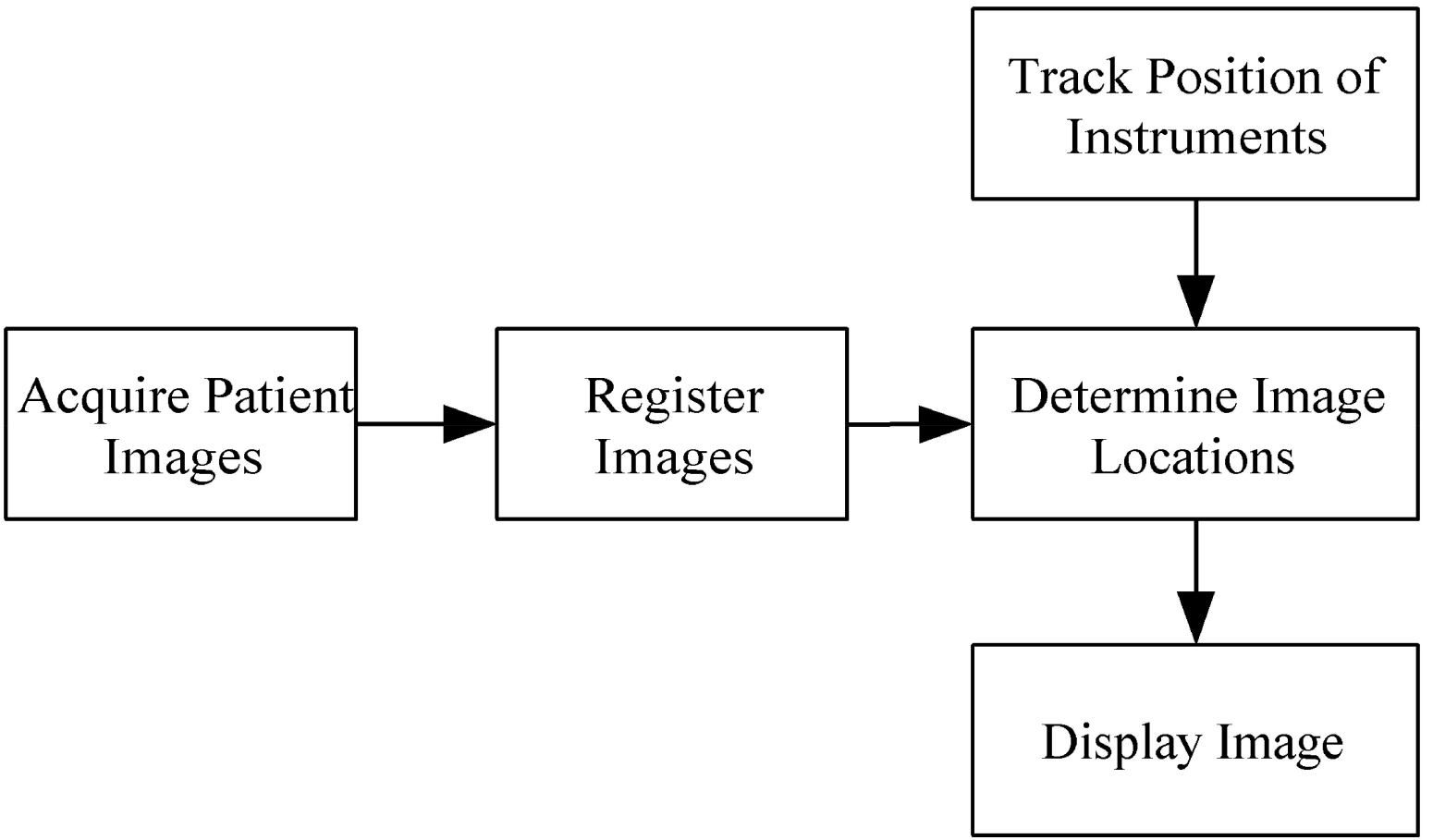
Source: CS133 Spring 2010 at UCLA (Kaplan)

Task Partition: Example

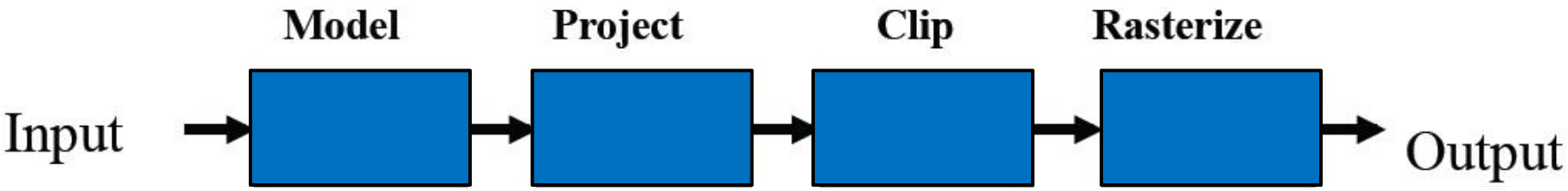


Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Task Partition: Another Example



Pipelining

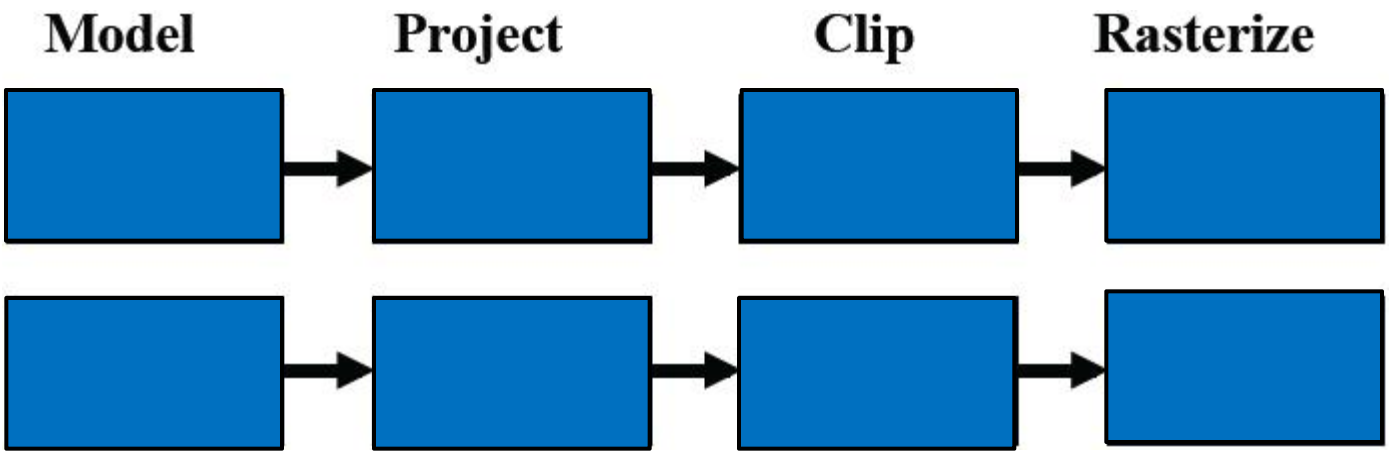


- Special kind of task decomposition
- "Assembly line" parallelism

- Example: 3D rendering in computer graphics

Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

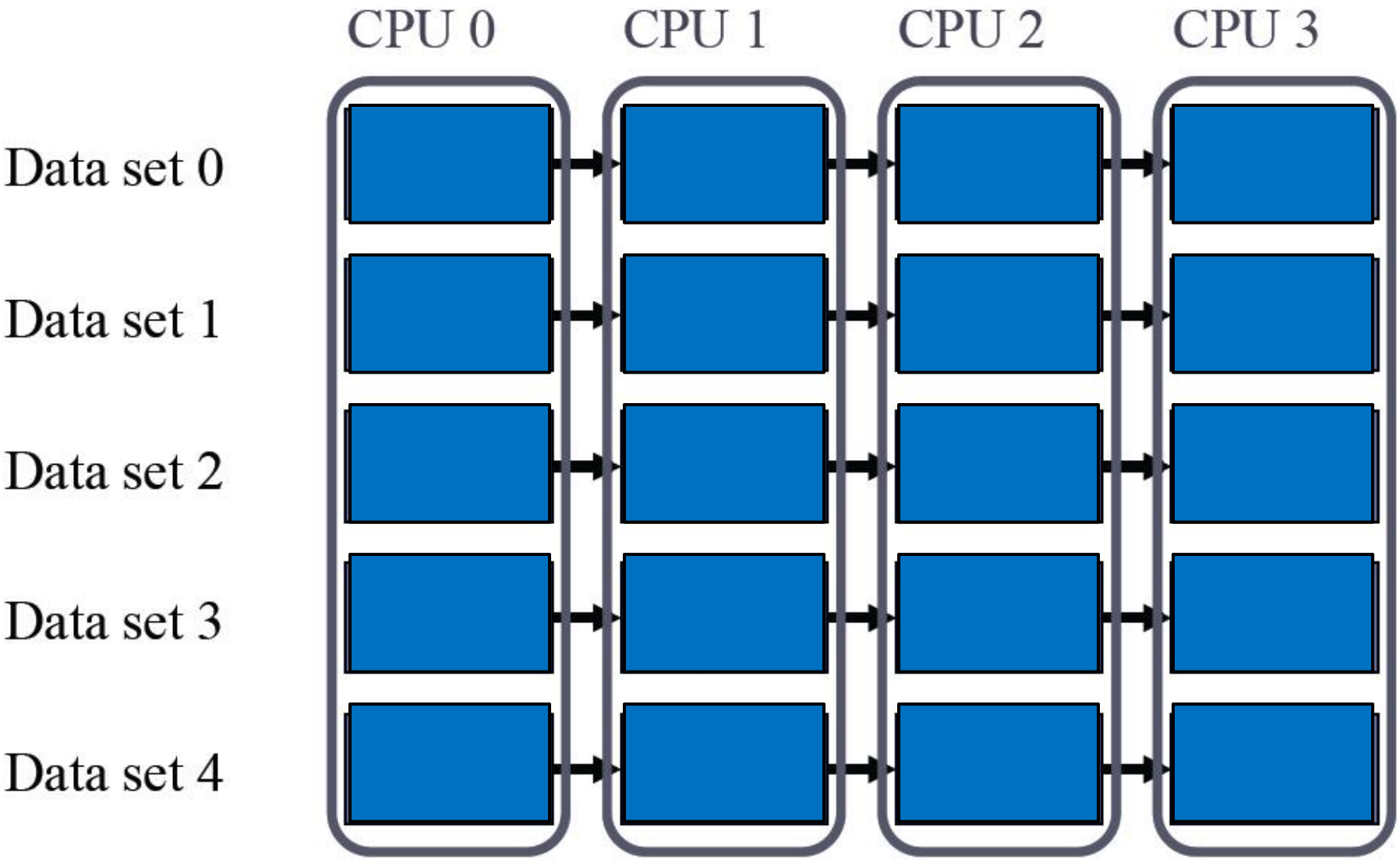
Processing Two Data Sets



The pipeline processes 2 data sets in 5 steps

Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Pipelining Five Data Sets



Source: Intel® Software College, copyright © 2006, Intel Corporation
Source: CS133 Spring 2010 at UCLA (Kaplan)

Speedup from Pipelining

- **In the previous examples,**
  - **Process 1 data set in 4 steps**
  - **Process 2 data sets in 5 steps**
  - **Process 5 data sets in 8 steps**
  - **Process N data sets in ?? steps**
- **Pipelining improves throughput, but not latency**

Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
  - If not, later design options may be too constrained
- Minimize redundant computations and redundant data storage
  - If not, the design may not work well when the size of the problem increases
- Primitive tasks roughly the same size
  - If not, it may be hard to balance work among the processors
- Number of tasks an increasing function of problem size
  - If not, it may be impossible to use more processors to solve large problem instances

Communication

- Determine values passed among tasks
  - *Task-channel graph*
- Local communication
  - Task needs values from a small number of other tasks
  - Create channels illustrating data flow
- Global communication
  - Significant number of tasks contribute data to perform a computation
  - Don't create channels for them early in design

Communication Checklist

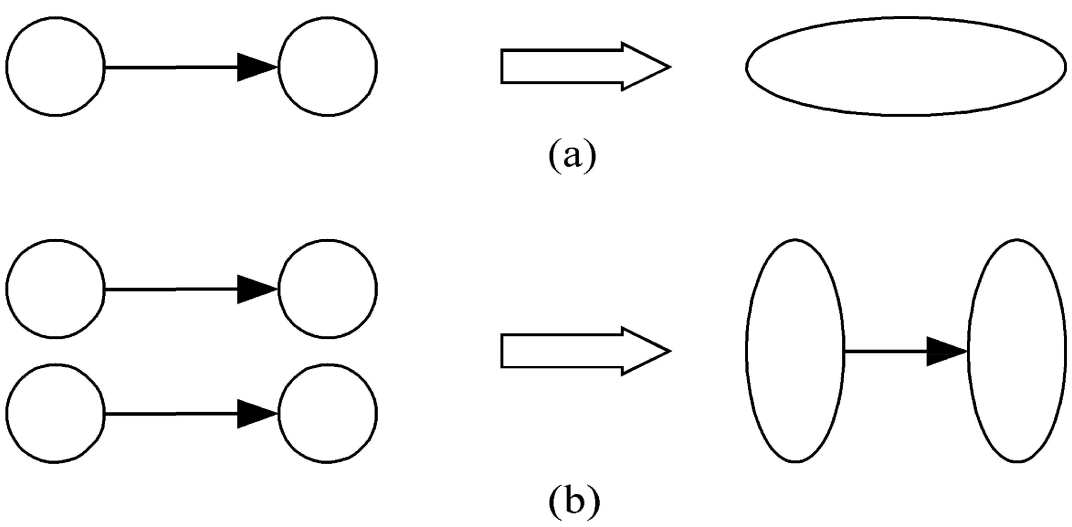
- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently

Agglomeration

- Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming
- In message-passing programming, goal often to create one agglomerated task per processor

Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task
- Combine groups of sending and receiving tasks



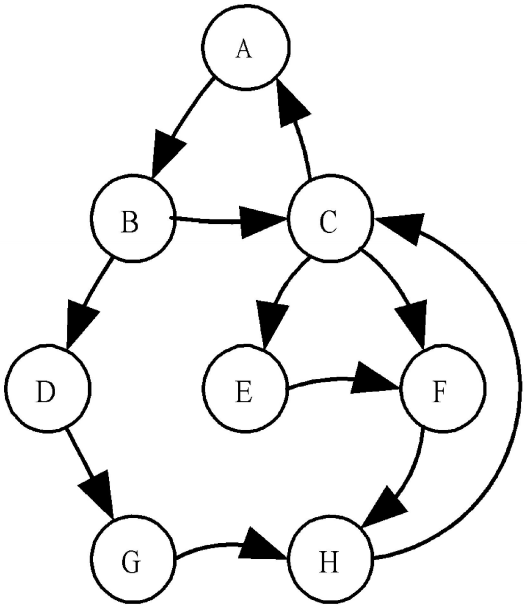
Agglomeration Checklist

- Locality of parallel algorithm has increased
- Replicated computations take less time than communications they replace
- Data replication doesn't affect scalability
- Agglomerated tasks have similar computational and communications costs
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

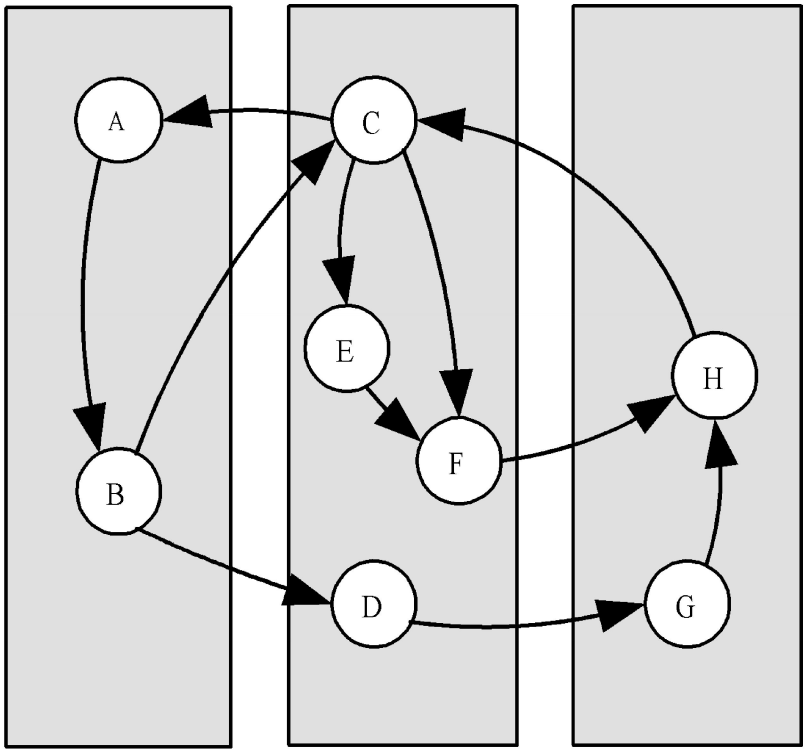
Mapping

- Process of assigning tasks to processors
  - Centralized multiprocessor: mapping done by operating system
  - Distributed memory system: mapping done by user
- Conflicting goals of mapping
  - Maximize processor utilization
  - Minimize interprocessor communication

Mapping Example



(a)



(b)

Optimal Mapping

- Finding optimal mapping is NP-hard
- Often rely on heuristics

Mapping Decision Tree

- **Static number of tasks**
  - **Structured communication**
    - **Constant computation time per task**
      - Agglomerate tasks to minimize communication
      - Create one task per processor
    - **Variable computation time per task**
      - Cyclically map tasks to processors
  - **Unstructured communication**
    - Use a static load balancing algorithm
- **Dynamic number of tasks**
  - **(next slide)**

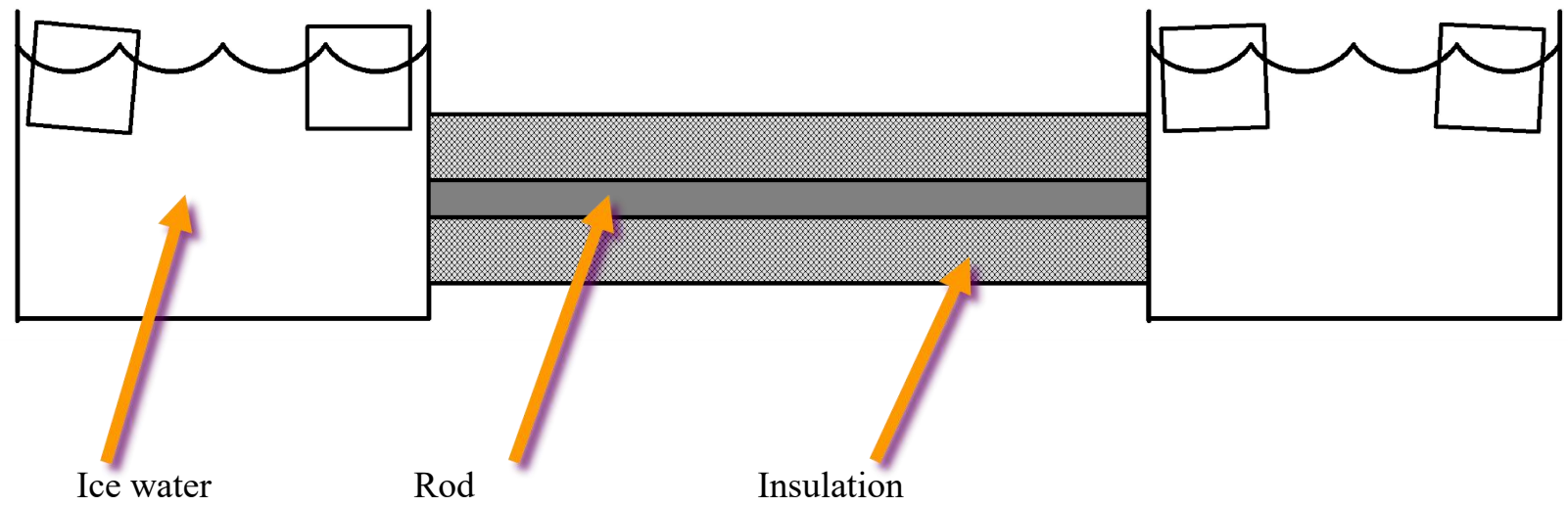
Mapping Decision Tree

- Static number of tasks
  - (previous slide)
- Dynamic number of tasks
  - Frequent communications between tasks
    - Use a dynamic load balancing algorithm
  - Many short-lived tasks
    - Use a runtime task-scheduling algorithm

Mapping Checklist

- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

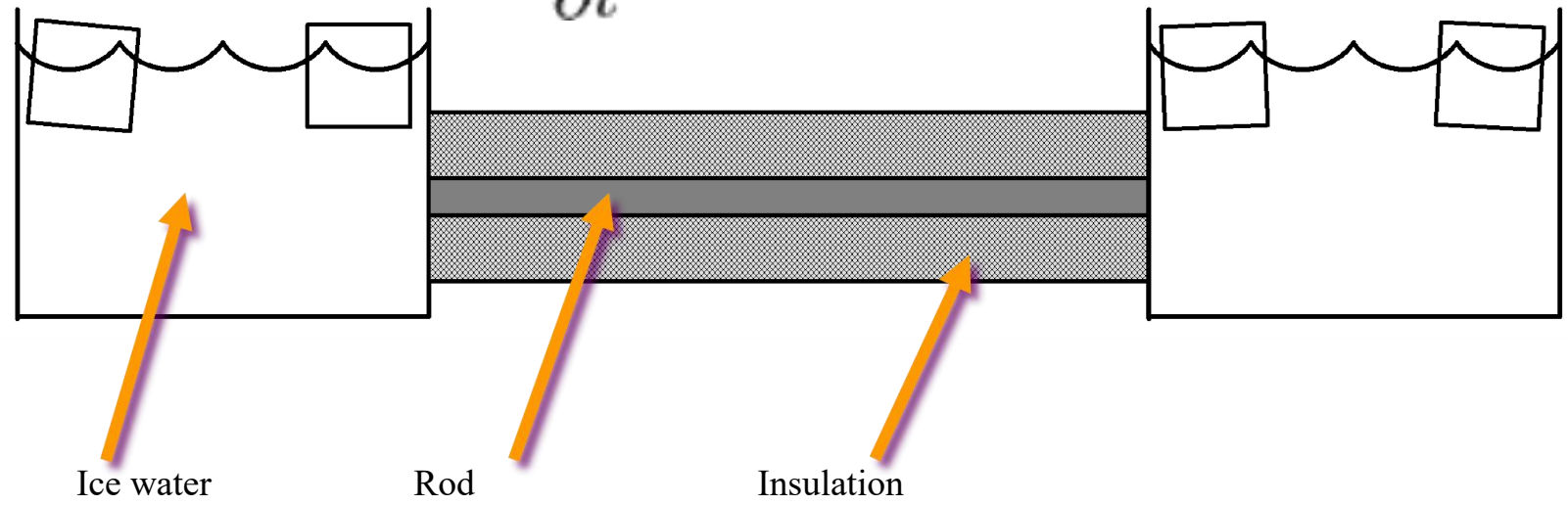
Case Study: Boundary Value Problem



A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

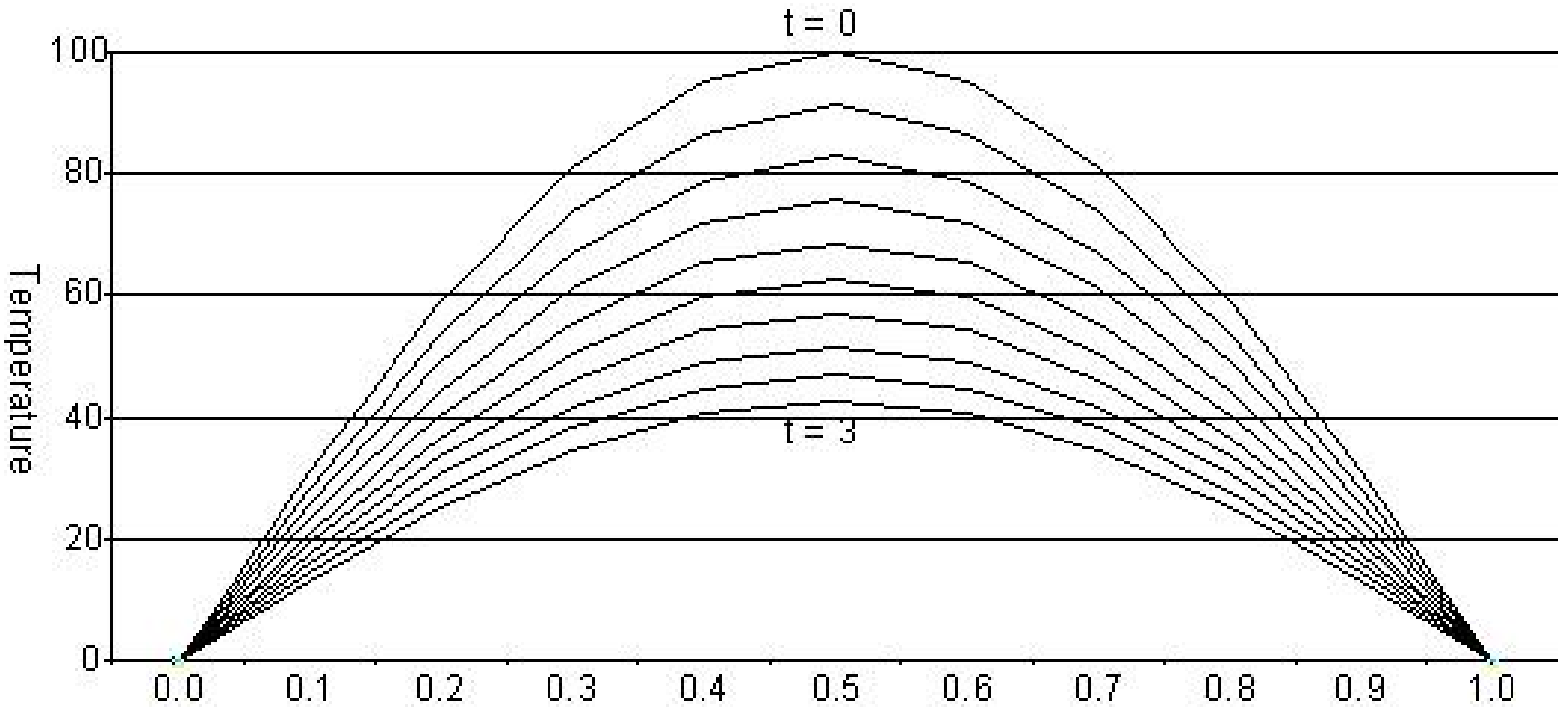
Problem Modeling

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$



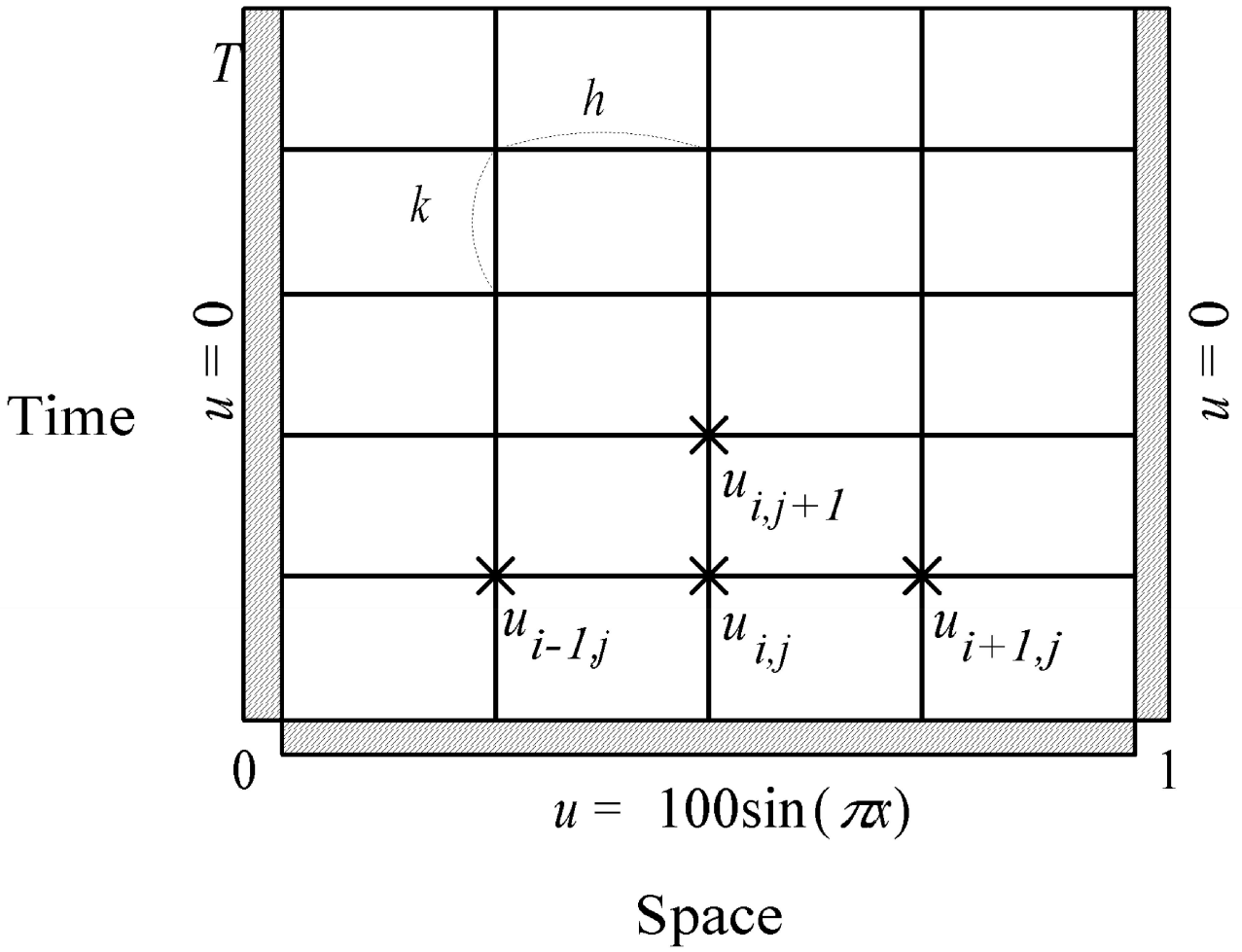
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Rod Cools as Time Progresses



A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finite Difference Approximation



A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition

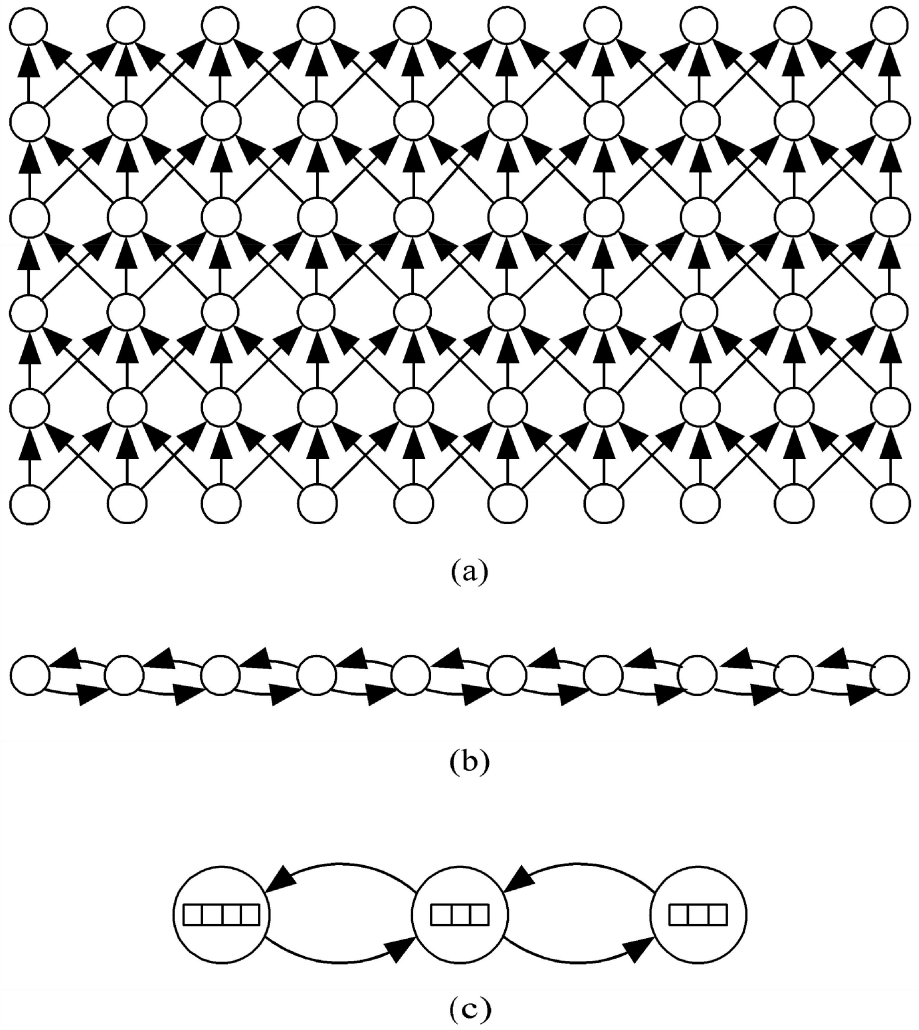
A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Communication

- **Identify communication pattern between primitive tasks**
- **Each interior primitive task has three incoming and three outgoing channels**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Agglomeration and Mapping



Agglomeration

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Summary: Architectures & Models

- **Parallel architectures**
  - SISD; SIMD; MISD
  - MIMD
    - Shared memory (UMA, NUMA)
    - Distributed memory
    - Hybrid
- **Programming models**
  - Shared address space model
  - Message passing model
  - Data parallel model

Summary: Design Methodologies

- **Incremental parallelization (esp. of existing programs)**
  - Study a sequential program (or code segment)
  - Look for bottlenecks & opportunities for parallelism
  - Try to keep all processors busy doing useful work
- **Design methodology for (new) parallel algorithms**
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping

Further Readings

- **Parallel Computer Memory Architectures**
  - https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##MemoryArch
- **Parallel Programming Models**
  - https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##Models
- **Designing Parallel Algorithms / Methodical Design**
  - https://www.mcs.anl.gov/~itf/dbpp/text/node14.html
  - (from the early days, but still illuminating)