Cornell University
Center for Advanced Computing

Introduction to Parallel Programming

January 14, 2015

What is Parallel Programming?

- Theoretically a very simple concept
  - Use more than one processor to complete a task
- Operationally much more difficult to achieve
  - Tasks must be independent
    - Order of execution can't matter
  - How to define the tasks
    - Each processor works on their section of the problem (functional parallelism)
    - Each processor works on their section of the data (data parallelism)
  - How and when can the processors exchange information

Why Do Parallel Programming?

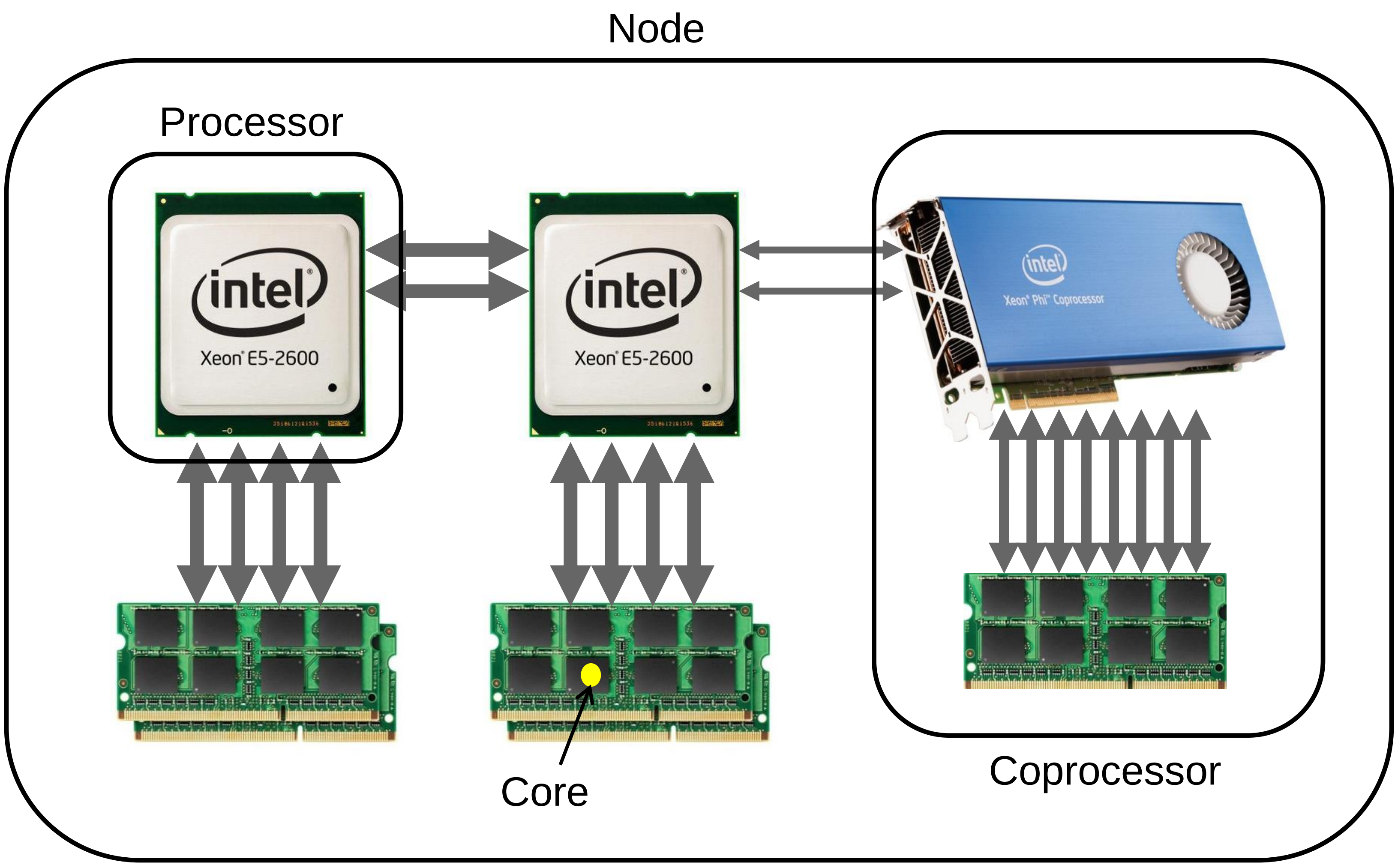
- Solve problems faster; 1 day is better than 30 days
- Solve bigger problems; model stress on a machine, not just one nut
- Solve problem on more datasets; find all max values for one month, not one day
- Solve problems that are too large to run on a single CPU
- Solve problems in real time

Is it worth it to go Parallel?

- Writing effective parallel applications is difficult!!
  - Load balancing is critical
  - Communication can limit parallel efficiency
  - Serial time can dominate
- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization? Is your problem really “large”?
  - Is there a library that does what you need (parallel FFT, linear system solving)
  - Will the code be used more than once?

Terminology

- **node:** a discrete unit of a computer system that typically runs its own instance of the operating system
  - Stampede has 6400 nodes
- **processor:** chip that shares a common memory and local disk
  - Stampede has two Sandy Bridge processors per node
- **core:** a processing unit on a computer chip able to support a thread of execution
  - Stampede has 8 cores per processor or 16 cores per node
- **coprocessor:** a lightweight processor
  - Stampede has a one Phi coprocessor per node with 61 cores per coprocessor
- **cluster:** a collection of nodes that function as a single resource


Node
Processor
intel
Xeon® E5-2600
intel
Xeon® E5-2600
intel
Xeon® Phi™ Coprocessor
Core
Coprocessor

Functional Parallelism

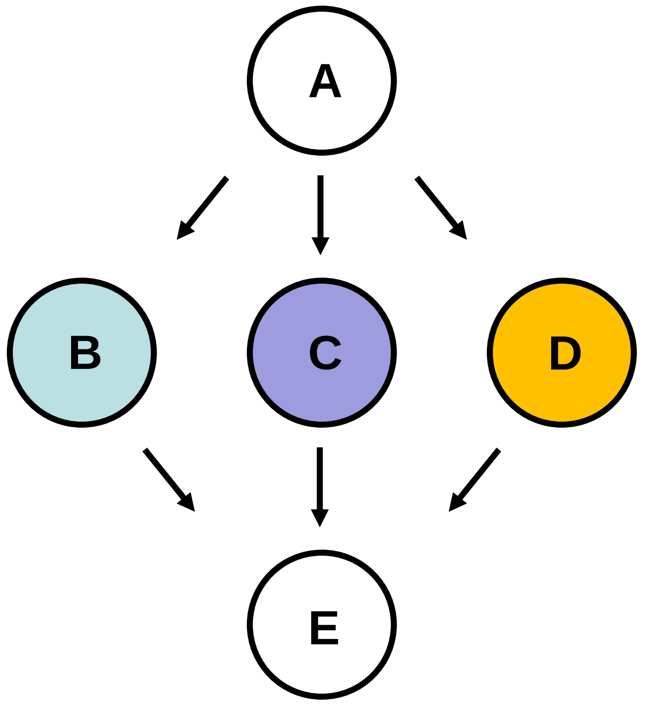
Definition: each process performs a different "function" or executes different code sections that are independent.

Examples:

2 brothers do yard work (1 edges & 1 mows)

8 farmers build a barn

- Commonly programmed with message-passing libraries



Data Parallelism

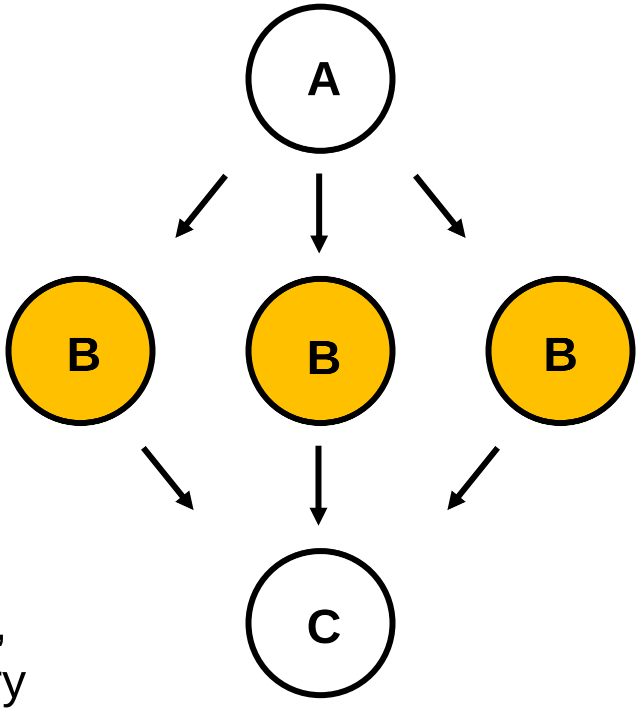
Definition: each process does the same work on unique and independent pieces of data

Examples:

2 brothers mow the lawn

8 farmers paint a barn

- Usually more scalable than functional parallelism
- Can be programmed at a high level with OpenMP, or at a lower level using a message-passing library like MPI or with hybrid programming.


$$\begin{tikzcd} & A \arrow[dl] \arrow[d] \arrow[dr] & \\ B \arrow[dr] & B \arrow[d] & B \arrow[dl] \\ & C & \end{tikzcd}$$

Embarrassing Parallelism
A special case of Data Parallelism

Definition: each process performs the same functions but do not communicate with each other, only with a “Master” Process. These are often called “Embarrassingly Parallel” codes.

Examples:

Independent Monte Carlo Simulations

ATM Transactions

Stampede has a special wrapper for submitting this type of job; see

https://www.xsede.org/news/-/news/item/5778

Flynn’s Taxonomy

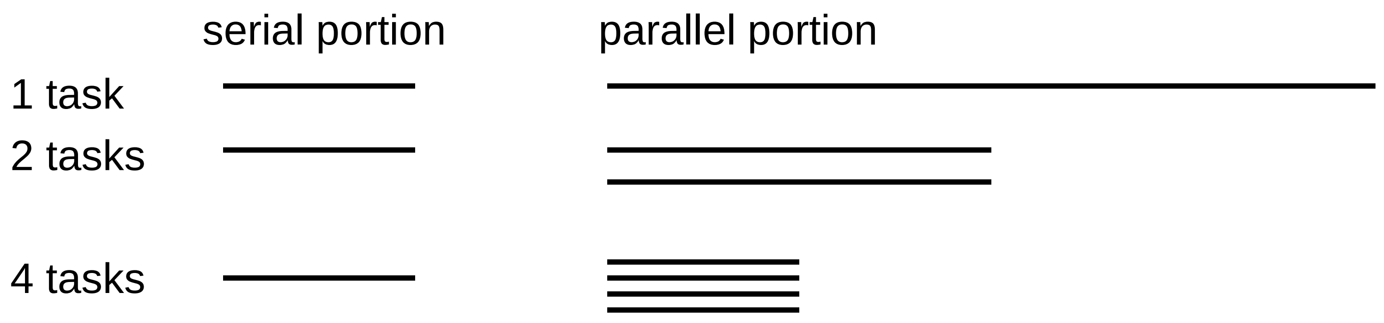
- Classification of computer architectures
- Based on # of concurrent instruction streams and data streams

	Single Instructio n	Multiple Instruction	Single Program	Multiple Program
Single Data	SISD (serial)	MISD (custom)		
Multiple Data	SIMD (vector) (GPU)	MIMD (superscalar)	SPMD (data parallel)	MPMD (task parallel)

Theoretical Upper Limits to Performance

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)

- Serial sections limit the parallel effectiveness



- Amdahl's Law states this formally

Amdahl's Law

- Amdahl's Law places a limit on the speedup gained by using multiple processors.
  - Effect of multiple processors on run time

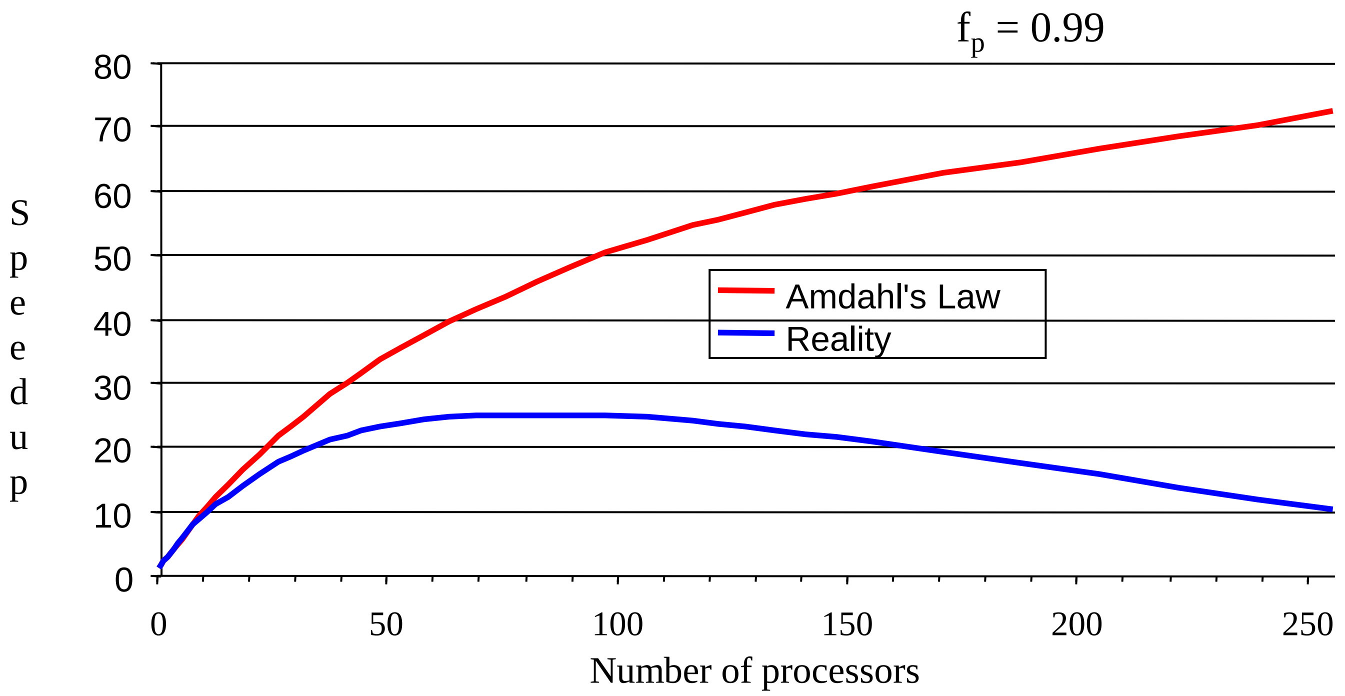
$$t_n = (f_p / N + f_s)t_1$$

  - where
    - f_s = serial fraction of the code
    - f_p = parallel fraction of the code
    - N = number of processors
    - t_1 = time to run on one processor

- Speed up formula: $S = 1 / (f_s + f_p / N)$
  - if $f_s = 0$ & $f_p = 1$, then $S = N$
  - If $N \rightarrow$ infinity: $S = 1/f_s$; if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors.

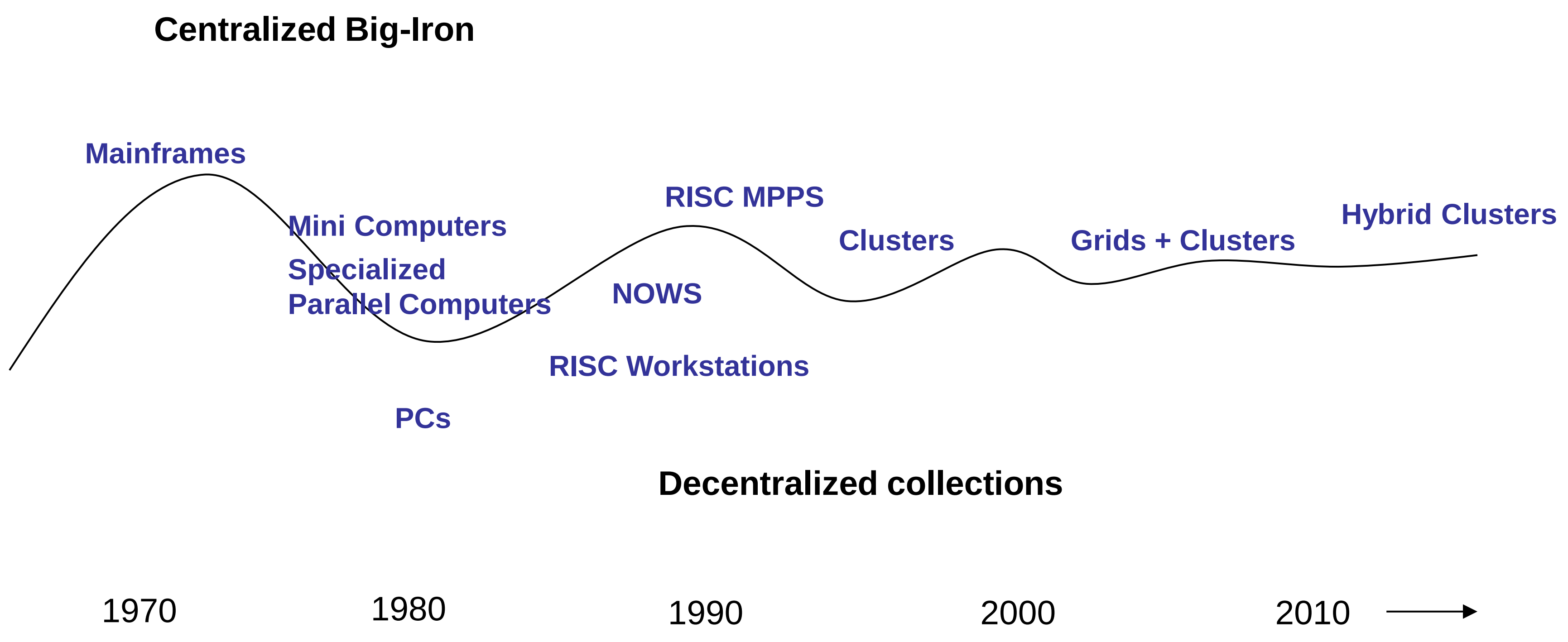
Practical Limits: Amdahl’s Law vs. Reality

- Amdahl’s Law shows a theoretical upper limit for speedup
- In reality, the situation is even worse than predicted by Amdahl’s Law due to:
  - – Load balancing (waiting)
  - – Scheduling (shared processors or memory)
  - – Communications
  - – I/O



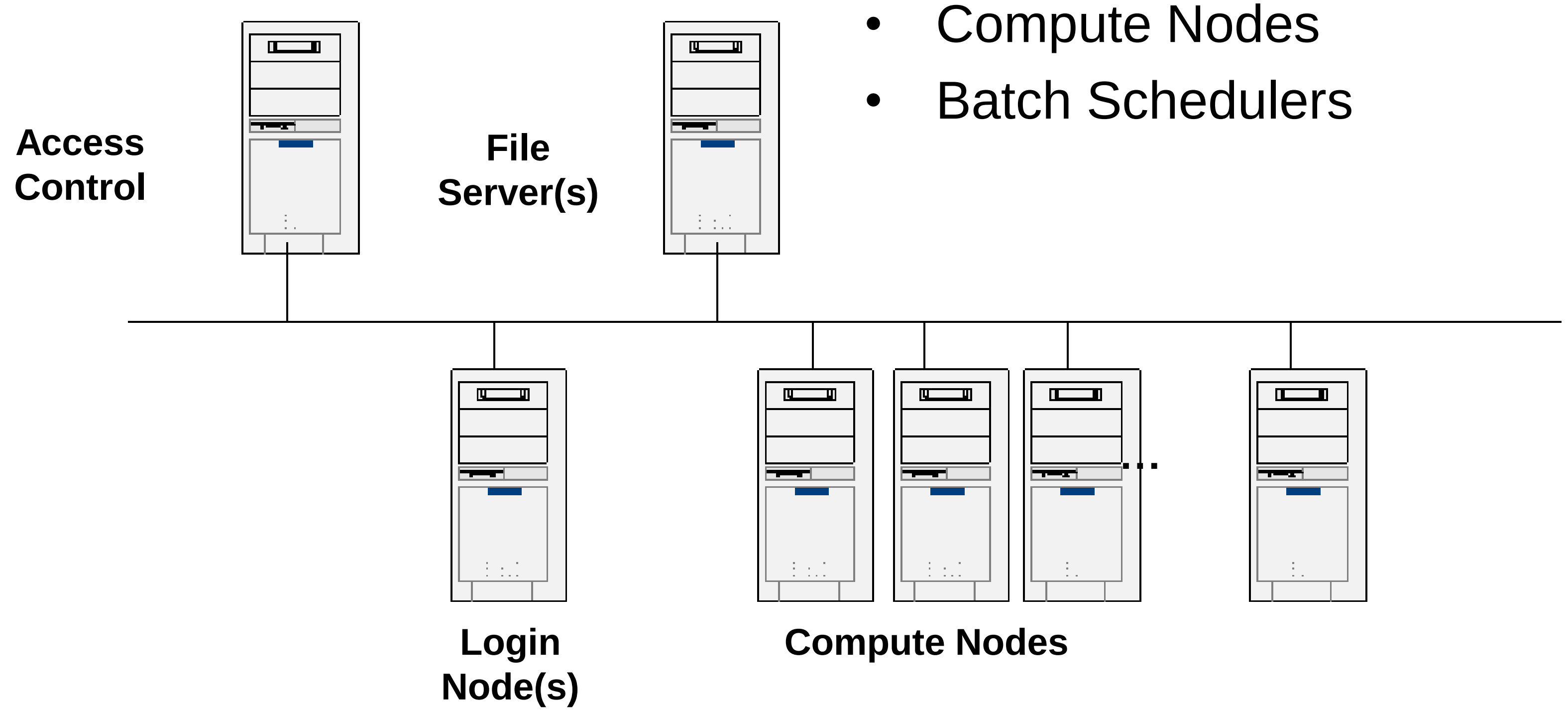
High Performance Computing Architectures

HPC Systems Continue to Evolve Over Time…



Cluster Computing Environment

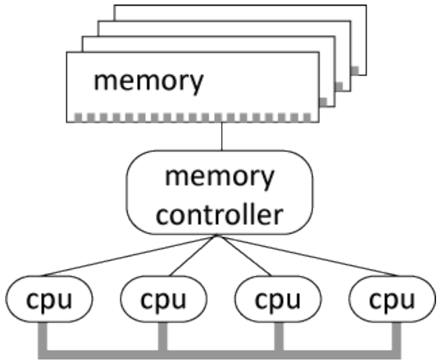
- Login Nodes
- File servers & Scratch Space
- Compute Nodes
- Batch Schedulers

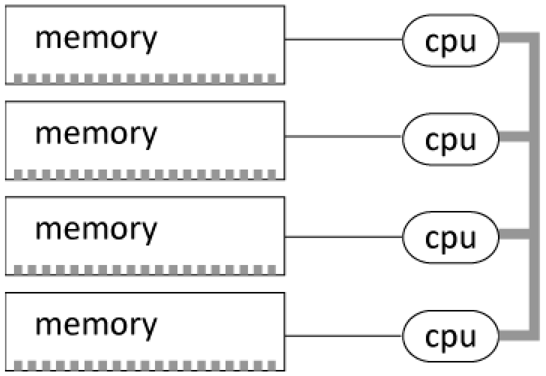


Types of Parallel Computers (Memory Model)

- Useful to classify modern parallel computers by their memory model
  - shared memory architecture
    memory is addressable by all cores and/or processors
  - distributed memory architecture
    memory is split up into separate pools, where each pool is addressable only by cores and/or processors on the same node
  - cluster
    mixture of shared and distributed memory; shared memory on cores in a single node and distributed memory between nodes
- Most parallel machines today are multiple instruction, multiple data (MIMD)

Shared and Distributed Memory Models





Shared memory: single address space. All processors have access to a pool of shared memory; easy to build and program, good price-performance for small numbers of processors; predictable performance due to uniform memory access (UMA).

Methods of memory access :
- Bus
- Crossbar

Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors. cc-NUMA enables larger number of processors and shared memory address space than SMPs; still easy to program, but harder and more expensive to build. (example: Clusters)

Methods of memory access :
- various topological interconnects

Programming Parallel Computers

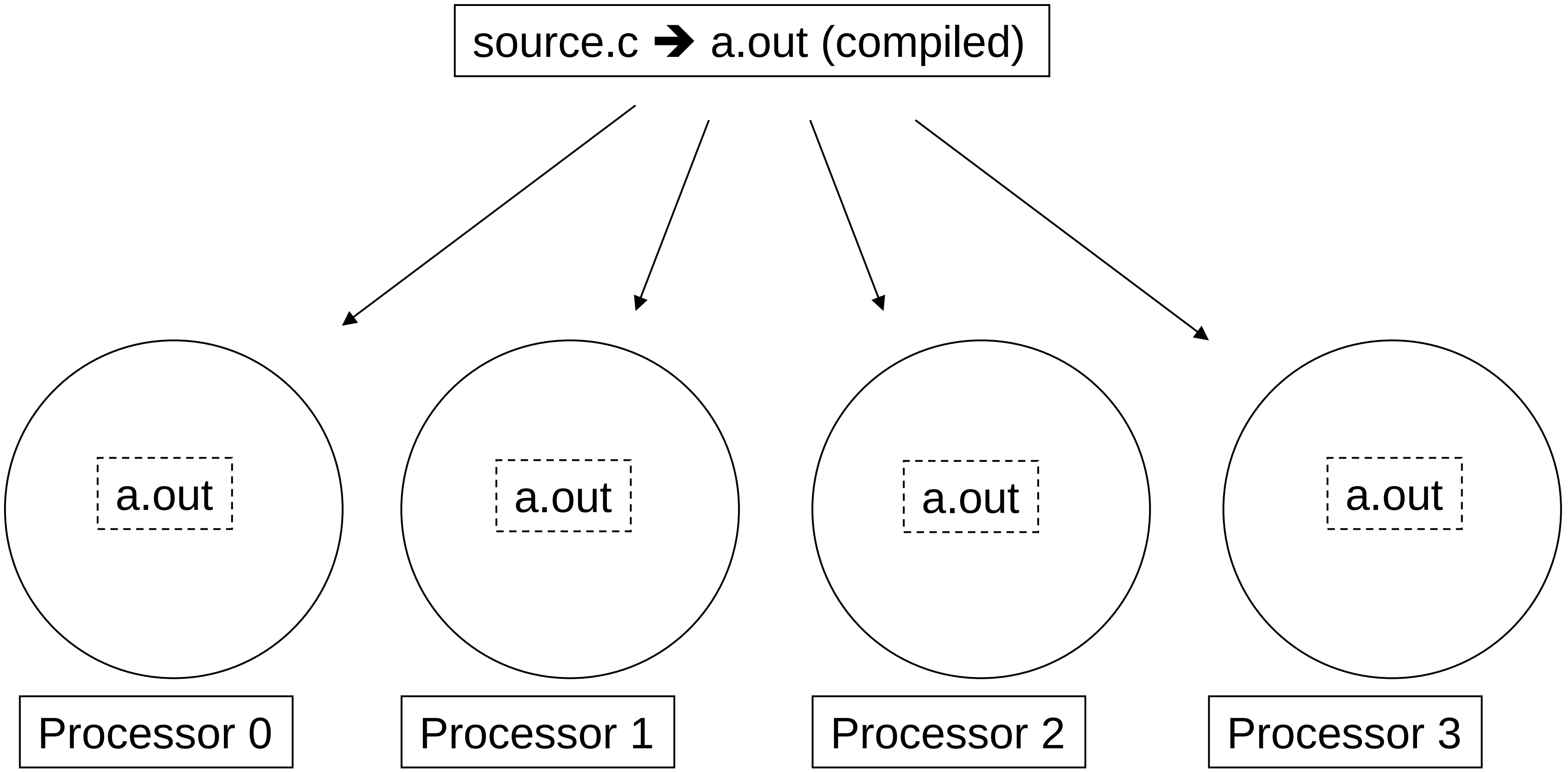
- Programming single-processor systems is (relatively) easy because they have a single thread of execution
- Programming shared memory systems can likewise benefit from the single address space
- Programming distributed memory systems is more difficult due to multiple address spaces and the need to access remote data
- Hybrid programming for distributed and shared memory is even more difficult, but gives the programmer much greater flexibility

Single Program, Multiple Data (SPMD)

SPMD:

- One source code is written
- Code can have conditional execution based on which processor is executing the copy
- All copies of code are started simultaneously and communicate and sync with each other periodically

SPMD Programming Model



Shared Memory Programming: OpenMP

- Shared memory systems have a single address space:
  - Applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - Application runs as a single process with multiple parallel threads
  - OpenMP is the standard for shared memory programming (compiler directives)
  - Vendors offer native compiler directives

Distributed Memory Programming: Message Passing Interface (MPI)

Distributed memory systems have separate pools of memory for each processor

- Application runs as multiple processes with separate address spaces
- Processes communicate data to each other using MPI
- Data must be manually decomposed
- MPI is the standard for distributed memory programming (library of subprogram calls)

Hybrid Programming

- Systems with multiple shared memory nodes

- Memory is shared at the node level, distributed above that:
  - Applications can be written to run on one node using OpenMP
  - Applications can be written using MPI
  - Application can be written using both OpenMP and MPI

References

- Virtual Workshop module:
  Parallel Programming Concepts and High-Performance Computing
- HPC terms glossary
- Jim Demmel, *Applications of Parallel Computers*. This set of lectures is an online rendition of Applications of Parallel Computers taught at U.C. Berkeley in Spring 2012. This online course is sponsored by the Extreme Science and Engineering Discovery Environment (XSEDE), and is only available through the XSEDE User Portal.
- Ian Foster, *Designing and Building Parallel Programs,* Addison-Wesley, 1995. http://www-unix.mcs.anl.gov/dbpp/ This book is a fine introduction to parallel programming and available online in its entirety. Some of the languages covered in later chapters, such as Compositional C++ and Fortran M, are more rare these days, but the concepts have not changed much since the book was written.