Introduction to Parallel Programming

David Lifka

lifka@cac.cornell.edu

May 23, 2011

What is Parallel Programming?

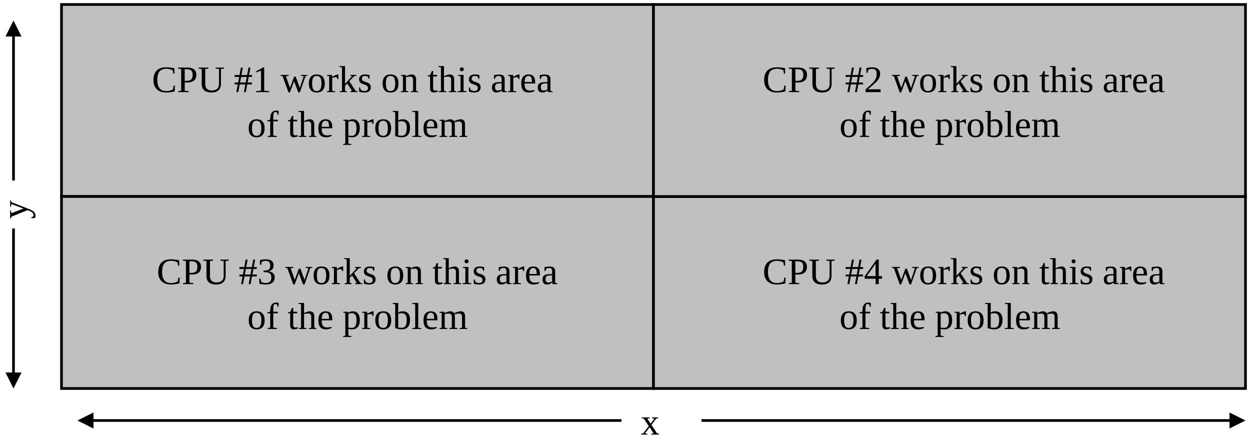
Using more than one processor or computer to complete a task

– Each processor works on its section of the problem (functional parallelism)

– Each processor works on its section of the data (data parallelism)

– Processors can exchange information

Grid of Problem to be solved



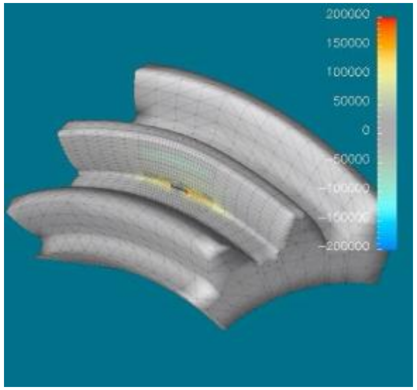
CPU #1 works on this area of the problem	CPU #2 works on this area of the problem
CPU #3 works on this area of the problem	CPU #4 works on this area of the problem

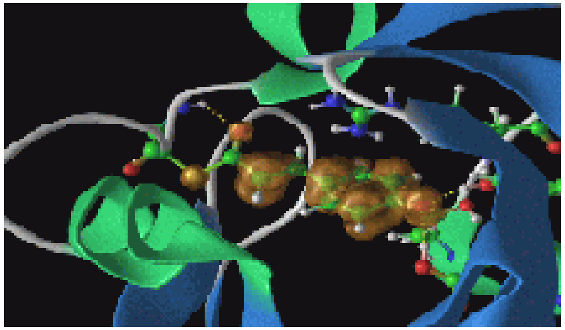
Why Do Parallel Programming?

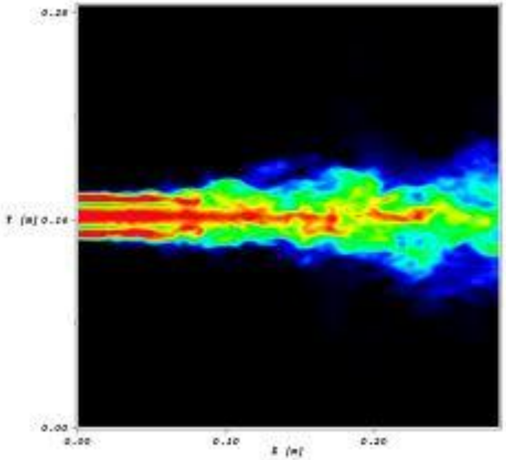
- Limits of single CPU computing
  - performance
  - available memory

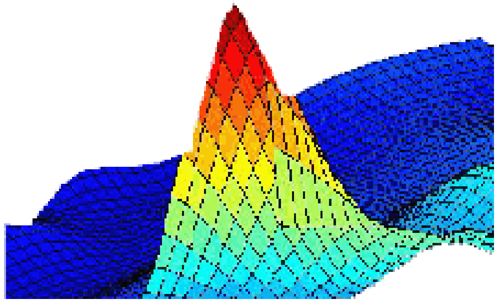
- Parallel computing allows one to:
  - solve problems that don’t fit on a single CPU
  - solve problems that can’t be solved in a reasonable time

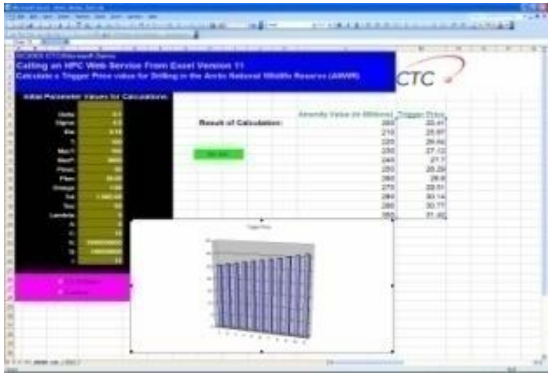
- We can solve…
  - larger problems
  - faster
  - more cases











CTC

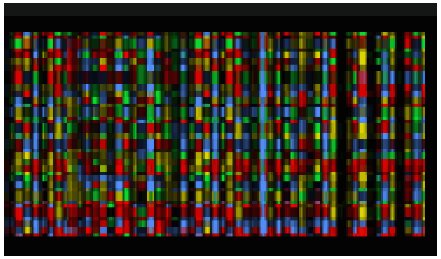
Calling an HPC Web Service From Excel Version 11

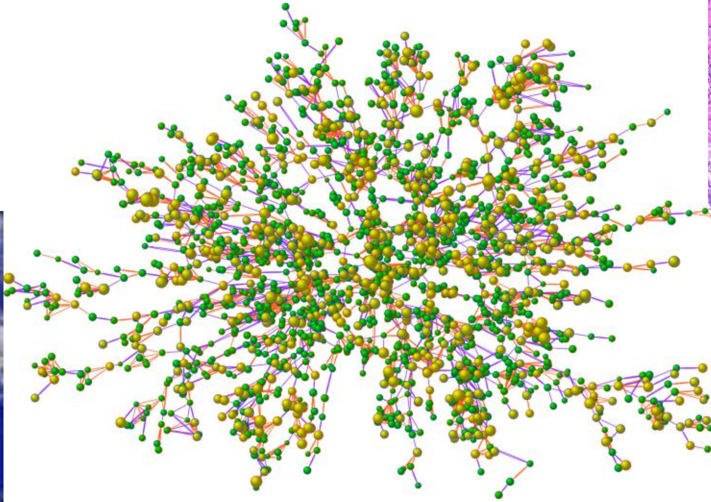
Calculate a Trigger Price value for Drilling in the Arctic National Wildlife Reserve (ANWR)

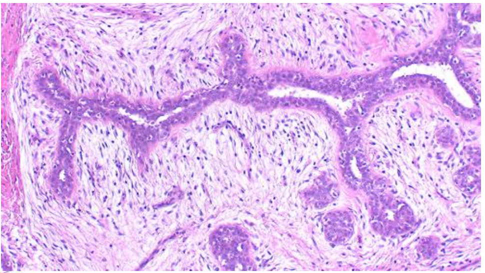
Initial Parameter Values for Calculations:

Result of Calculation:

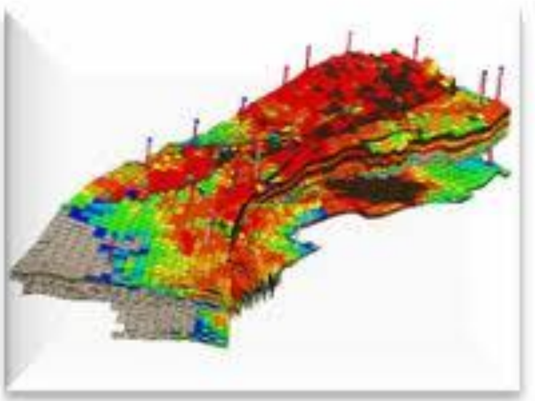
Amenity Value (in Millions)	Trigger Price
200	22.41
210	25.97
220	26.46
230	27.12
240	27.7
250	28.29
260	28.8
270	29.51
280	30.14
290	30.77
300	31.40







MTSAT 紅外線雲圖 7/10 10:00



Terminology (1)

- **serial** code is a single thread of execution working on a single data item at any one time

- **parallel** code has more than one thing happening at a time. This could be
  - A single thread of execution operating on multiple data items simultaneously
  - Multiple threads of execution in a single executable
  - Multiple executables all working on the same problem
  - Any combination of the above

- **task** is the name we use for an instance of an executable. Each task has its own virtual address space and may have multiple threads.

Terminology (2)

- **node:** a discrete unit of a computer system that typically runs its own instance of the operating system
- **core:** a processing unit on a computer chip that is able to support a thread of execution; can refer either to a single core or to all of the cores on a particular chip
- **cluster:** a collection of machines or nodes that function in someway as a single resource.
- **grid:** the software stack designed to handle the technical and social challenges of sharing resources across networking and institutional boundaries. grid also applies to the groups that have reached agreements to share their resources.

Types of Parallelism

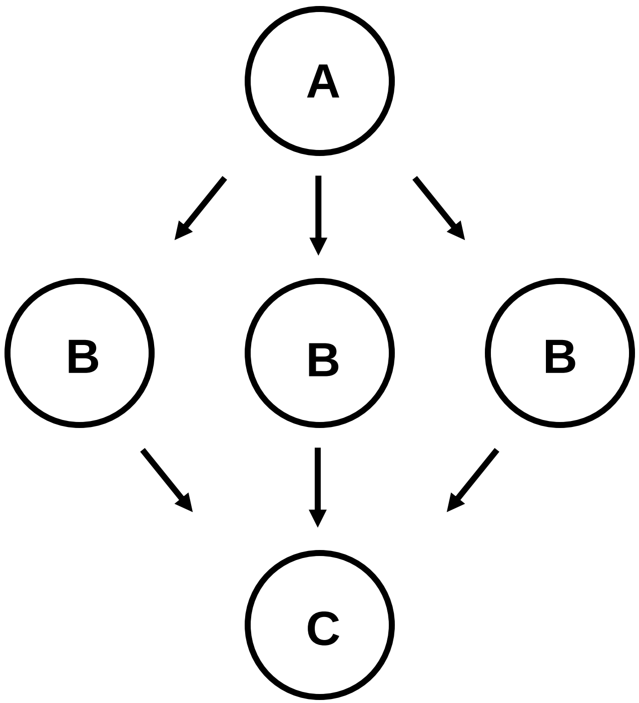
Data Parallelism

Definition: when independent tasks can apply the same operation to different elements of the data set at the same time.

Examples:

2 brothers mow the lawn

8 farmers paint a barn

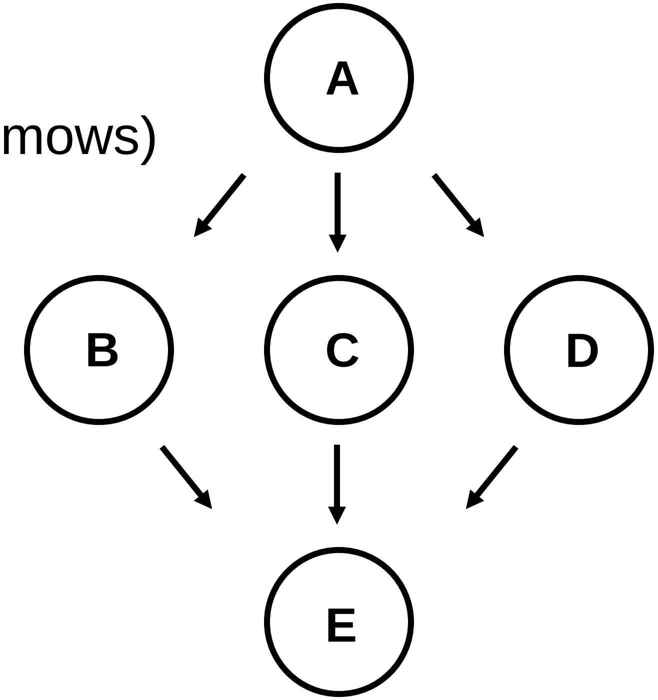


Functional Parallelism

Definition: when independent tasks can apply different operations to different data elements at the same time.

Examples:

- 2 brothers do yard work (1 edges & 1 mows)
- 8 farmers build a barn



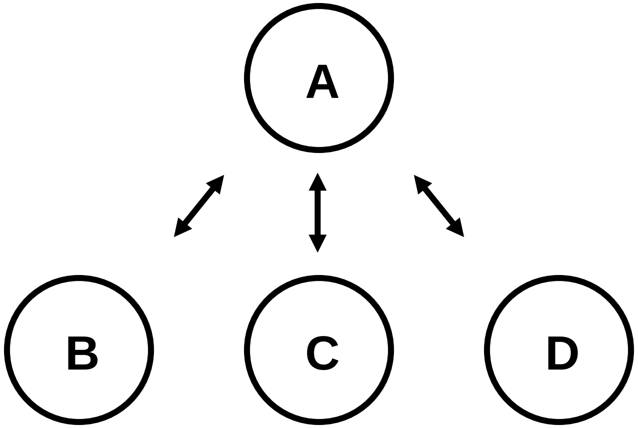
Task Parallelism

Definition: independent tasks perform functions but do not communicate with each other, only with a “Master” Process. These are often called “Embarrassingly Parallel”.

Examples:

Independent Monte Carlo Simulations

ATM Transactions



Pipeline Parallelism

Definition: Each Stage works on a part of a solution. The output of one stage is the input of the next. (Note: This works best when each stage takes the same amount of time to complete)

Examples: Assembly lines, Computing partial sums

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
A	i	i+1	i+2	i+3	i+4	i+5	i+6			
B		i	i+1	i+2	i+3	i+4	i+5	i+6		
C			i	i+1	i+2	i+3	i+4	i+5	i+6	
				i	i+1	i+2	i+3	i+4	i+5	i+6

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Effect of multiple processors on run time

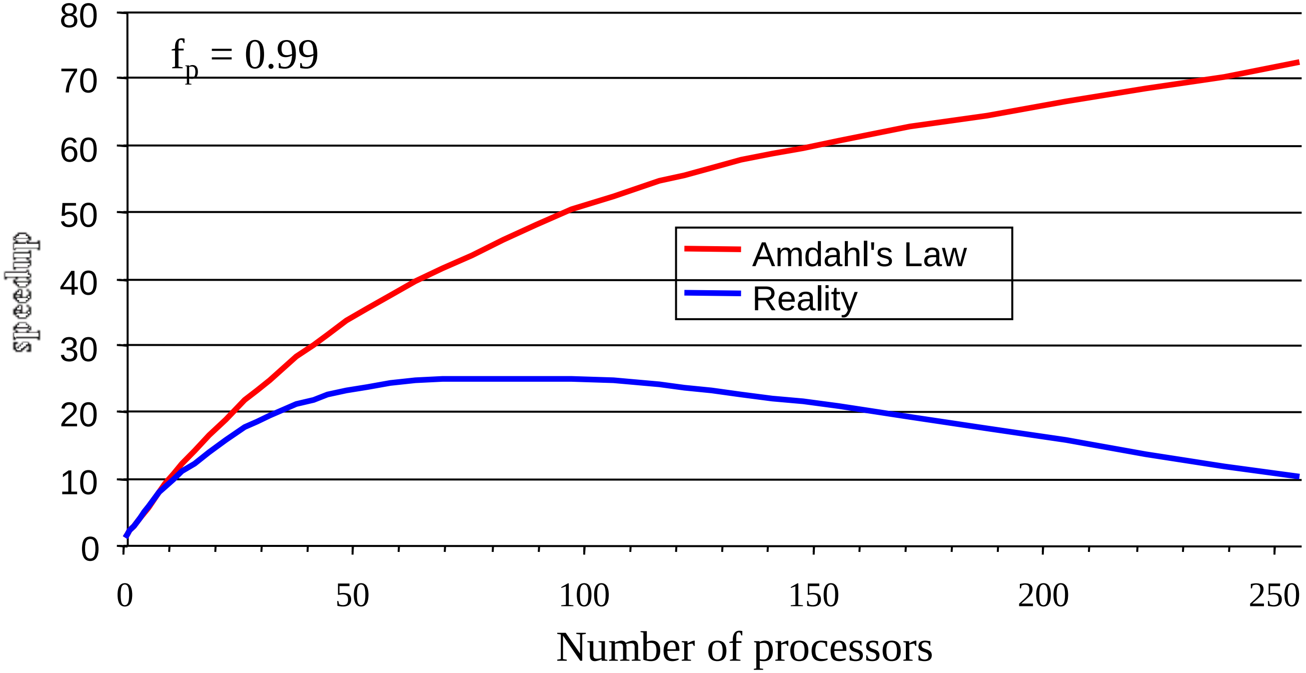
$$t_n = (f_p / N + f_s) t_1$$

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

Practical Limits: Amdahl’s Law vs. Reality

- Amdahl’s Law shows a theoretical upper limit || 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

S



More Terminology

- **synchronization**: the temporal coordination of parallel tasks. It involves waiting until two or more tasks reach a specified point (a sync point) before continuing any of the tasks.

- **parallel overhead**: the amount of time required to coordinate parallel tasks, as opposed to doing useful work, including time to start and terminate tasks, communication, move data.

- **granularity**: a measure of the ratio of the amount of computation done in a parallel task to the amount of communication.
  – fine-grained (very little computation per communication-byte)
  – coarse-grained (extensive computation per communication-byte).

Performance Considerations

Computationally or Data Intensive Applications

Have Critical Sections or “Hot Spots” where a majority of the application time is spent

Tightly Coupled Parallel Approaches

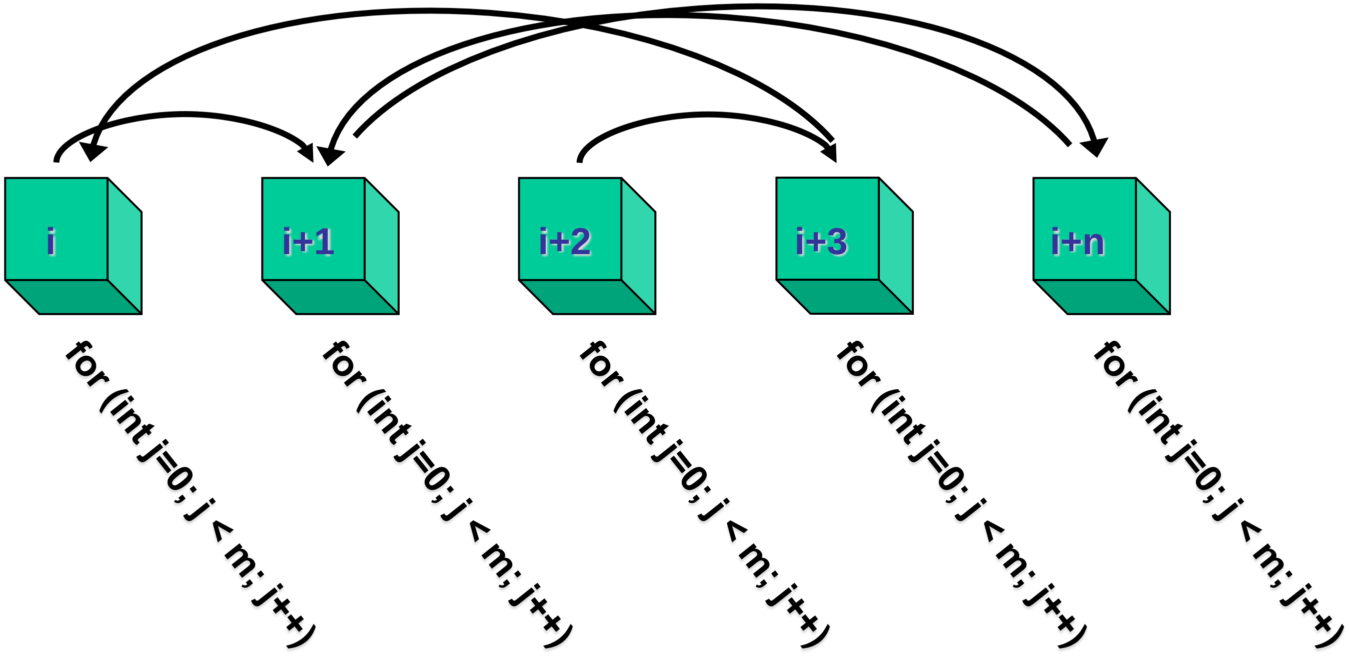
Parallel tasks must exchange data during the computation

Loosely Coupled Parallel Approaches

Parallel tasks can complete independent of each other

```
for (int i=0; i < n; i++)
 {
  for (int j=0; j < m; j++)
   {
    //Perform Calculation Here
   } // for j
 } // for i
```

for (int i=0; i < n; i++)



i

for (int j=0; j < m; j++)

i+1

for (int j=0; j < m; j++)

i+2

for (int j=0; j < m; j++)

i+3

for (int j=0; j < m; j++)

i+n

for (int j=0; j < m; j++)

Workers

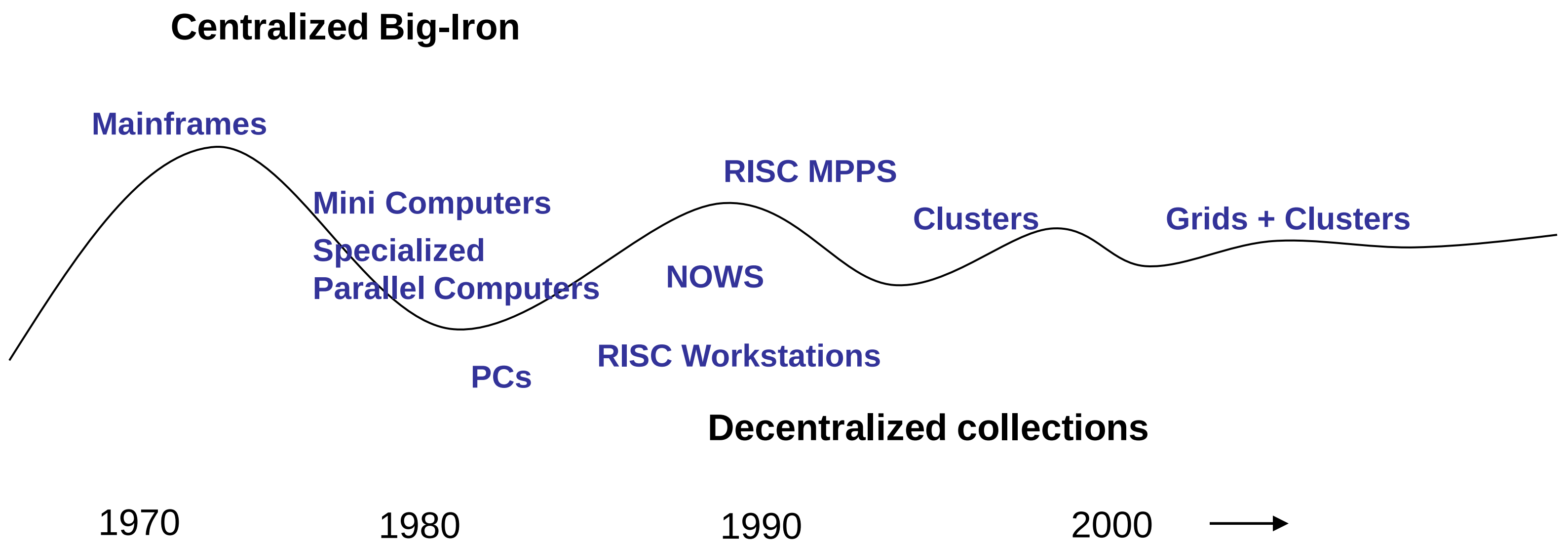
Is it really worth it to go Parallel?

- Writing effective parallel applications is difficult!!
  - Load balance is important
  - 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?

High Performance Computing Architectures

HPC Systems Continue to Evolve Over Time…



Centralized Big-Iron

Mainframes

Mini Computers

Specialized
Parallel Computers

PCs

RISC Workstations

NOWS

RISC MPPS

Clusters

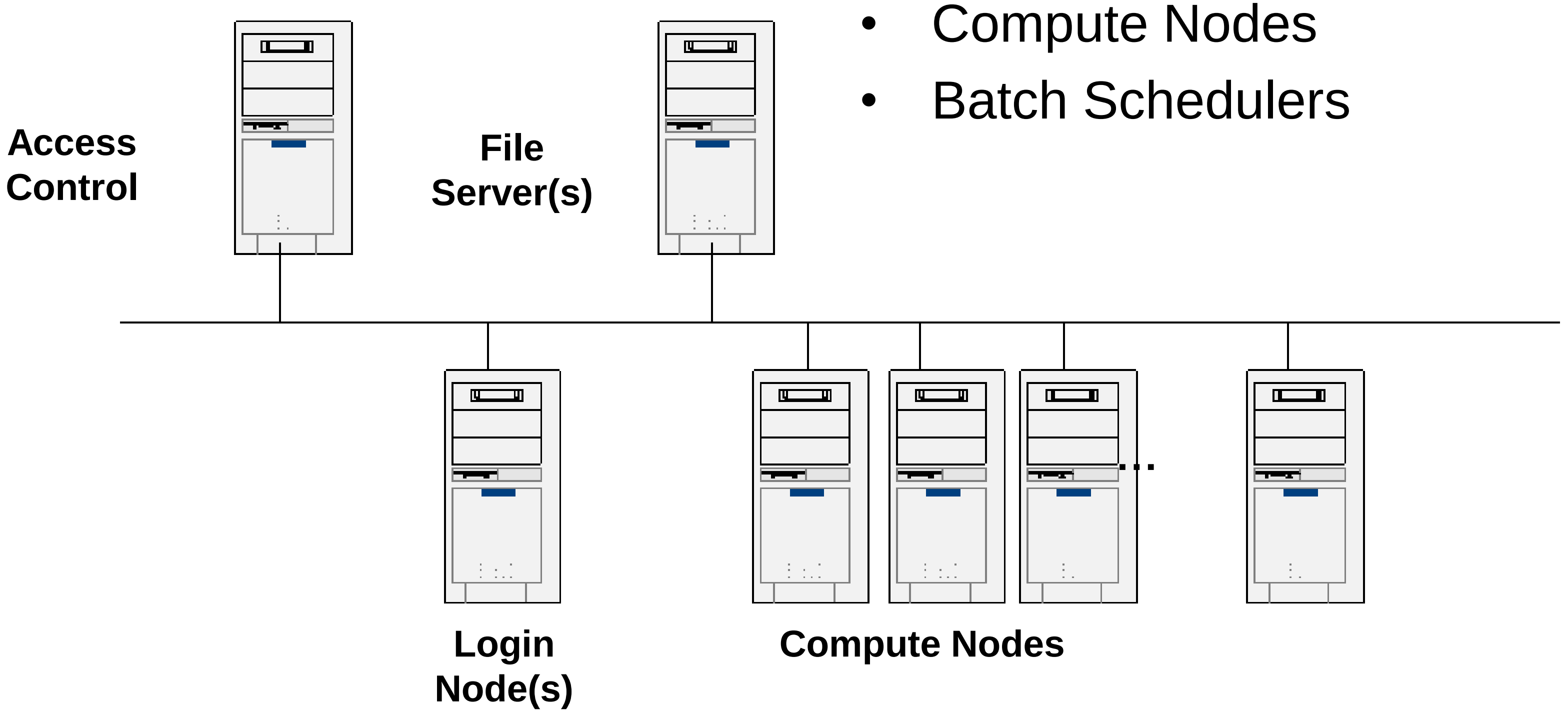
Grids + Clusters

Decentralized collections

1970 1980 1990 2000 →

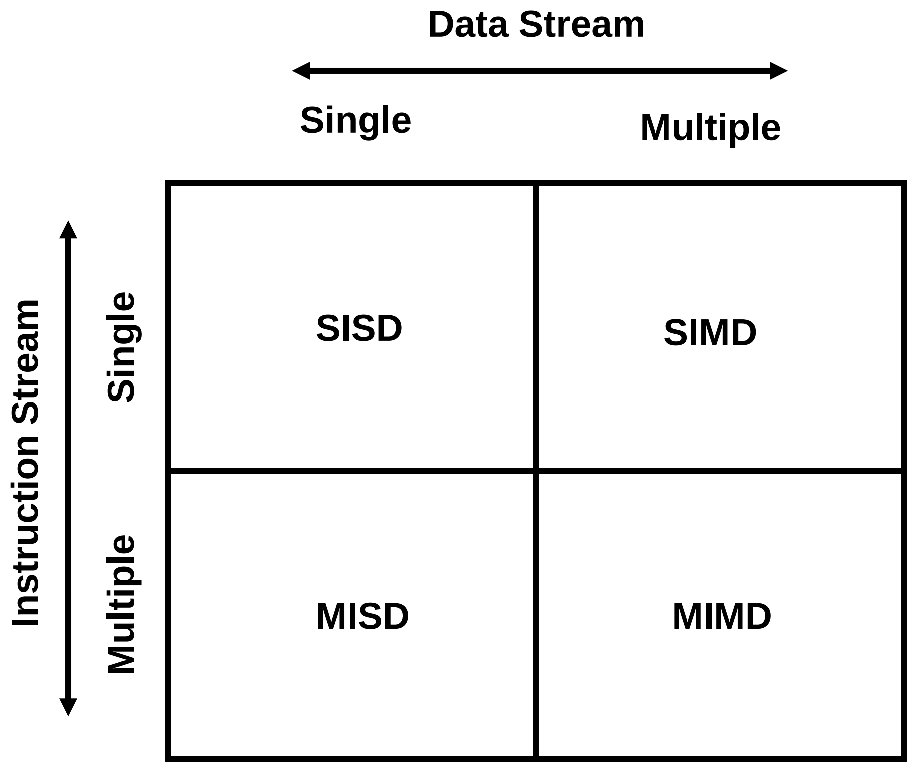
Cluster Computing Environment

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



Flynn’s Taxonomy

Classification Scheme for Parallel Computers



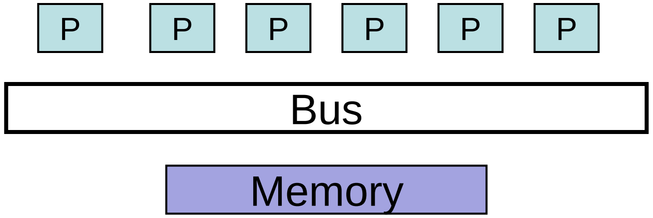
		Data Stream	
		Single	Multiple
Instruction Stream	Single	SISD	SIMD
	Multiple	MISD	MIMD

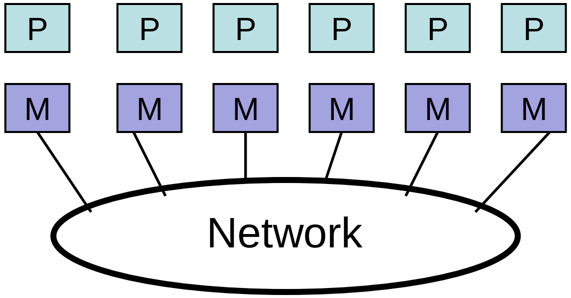
Types of Parallel Computers (Memory Model)

- Nearly all parallel machines these days are multiple instruction, multiple data (MIMD)

- A much more useful way to classify modern parallel computers is by their memory model
  - shared memory
  - distributed memory

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 UMA .(example: SGI Altix)

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

Shared Memory vs. Distributed Memory

- Tools can be developed to make any system appear to look like a different kind of system
  - distributed memory systems can be programmed as if they have shared memory, and vice versa
  - such tools do not produce the most efficient code, but might enable portability

- HOWEVER, the most natural way to program any machine is to use tools and languages that express the algorithm explicitly for the architecture.

Programming Parallel Computers

- Programming single-processor systems is (relatively) easy because they have a single thread of execution and a single address space.

- *Programming shared memory systems can benefit from the single* address space

- *Programming distributed memory systems is the most difficult due to* multiple address spaces and need to access remote data

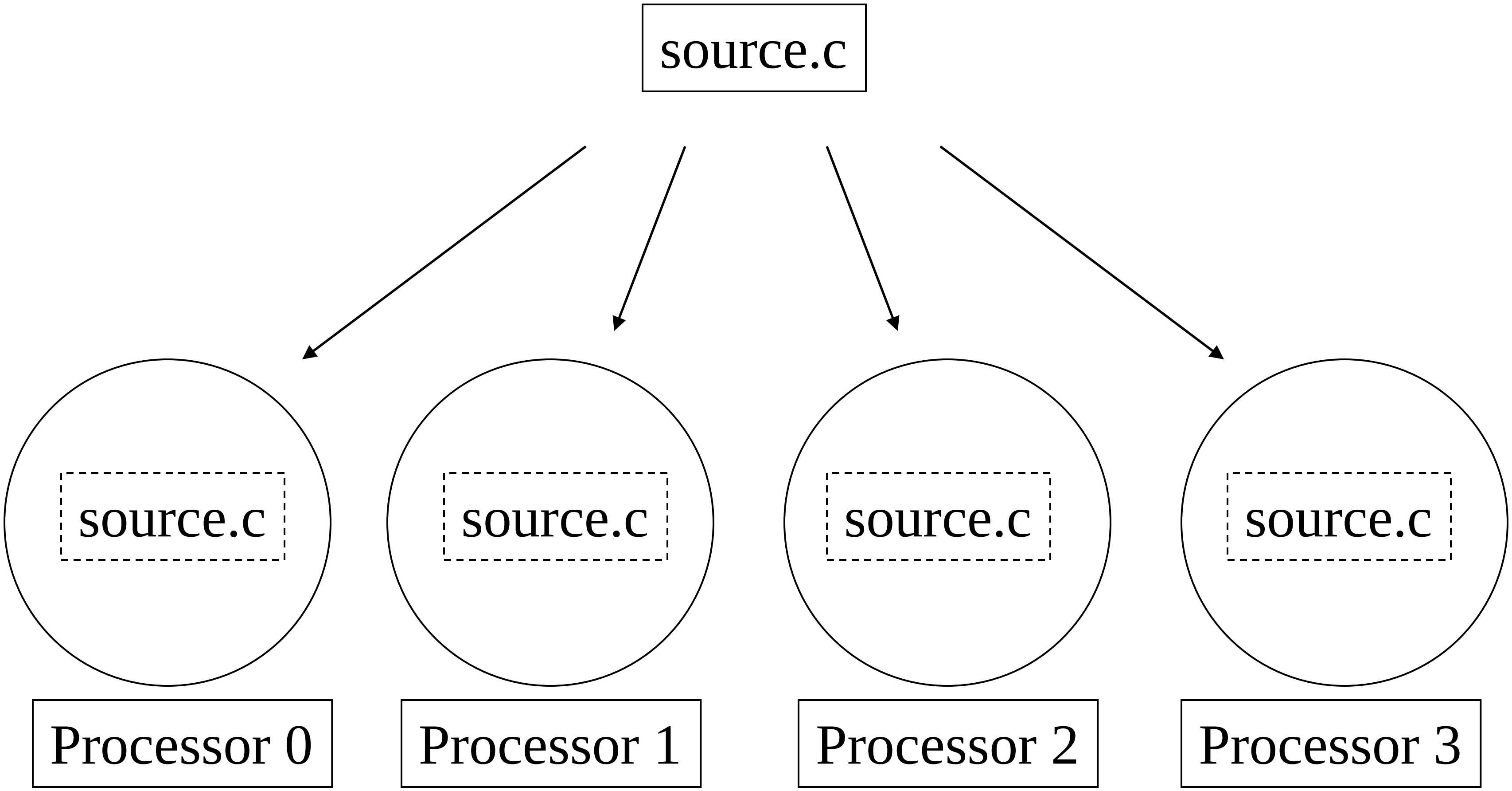
- Both shared memory and distributed memory parallel computers can be programmed in a data parallel, SIMD fashion and they also can perform independent operations on different data (MIMD) and implement task parallelism.

Single Program, Multiple Data (SPMD)

SPMD: dominant programming model for shared and distributed memory machines.

- 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



Questions?