Chapter 6

Parallel Processors from Client to Cloud

- A Very Quick Look

Section 6.1

Introduction

Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

What We've Already Covered

- §2.10: Parallelism and Instructions
  - Synchronization
- §3.6: Parallelism and Computer Arithmetic
  - Subword Parallelism
- §4.10: Parallelism and Advanced Instruction-Level Parallelism
- §5.10: Parallelism and Memory Hierarchies
  - Cache Coherence

Section 6.2

The difficulty of creating parallel processing programs

Parallel Programming

- Parallel software is the problem
- Difficult to write software that use multiple processors to complete one task faster
  - Gets worse as number of processors increase
- Why difficult?
  - Partitioning, load balancing
  - Coordination (synchronization, scheduling)
  - Communications overhead

Parallel Programming

- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor
- Sequential programming is simpler and
- Uniprocessor design techniques exploit instruction-level parallelism without involvement of programmers
  - Pipelining
  - Superscalar
  - Out-of-order execution

Section 6.3

SISD, MIMD, SIMD, SPMD, and Vector

Instruction and Data Streams

- An old classification of parallel hardware

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

MIMD and SIMD

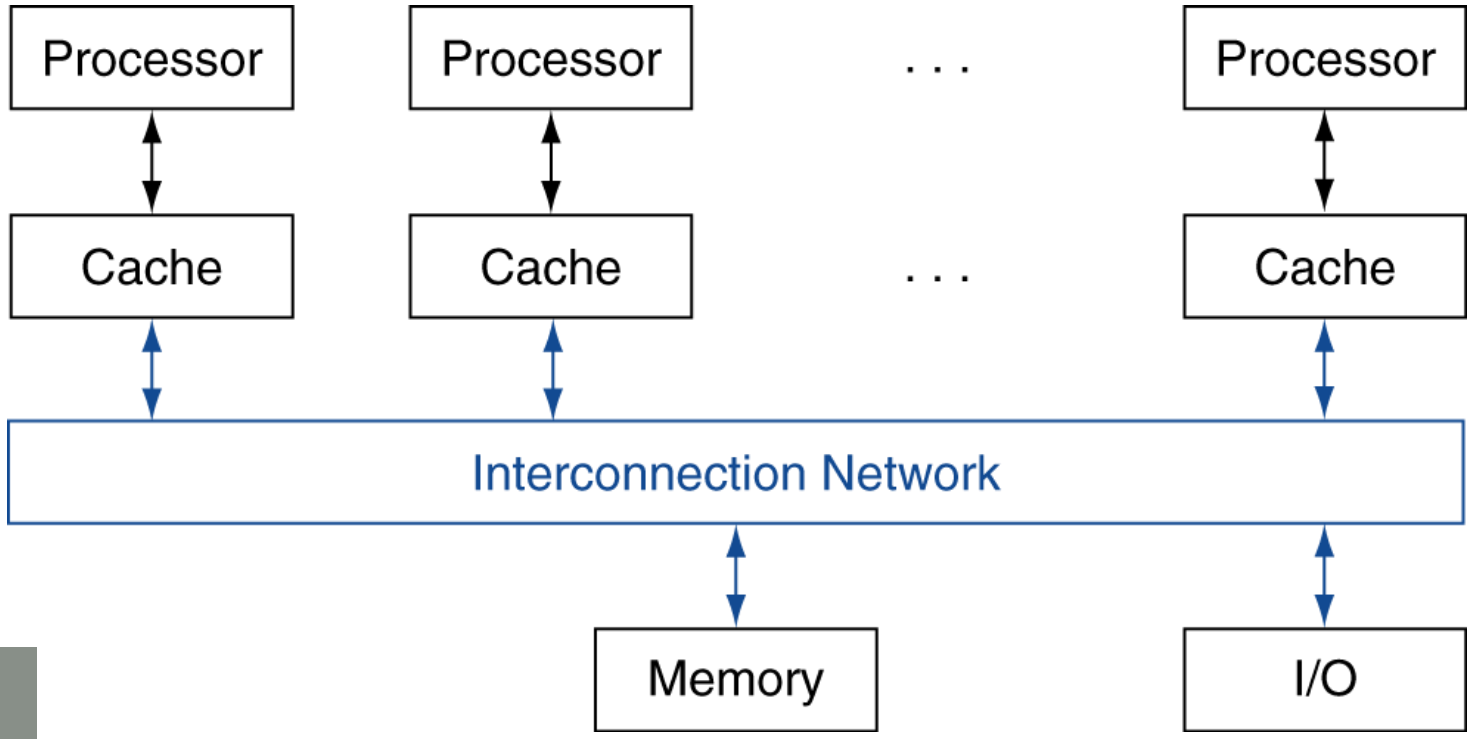
- MIMD: our focus
  - Two flavours
    - Shared memory multiprocessors (SMPs)
    - Clusters
- Let’s examine SIMD first

Section 6.5

Multicore and other shared memory multiprocessors

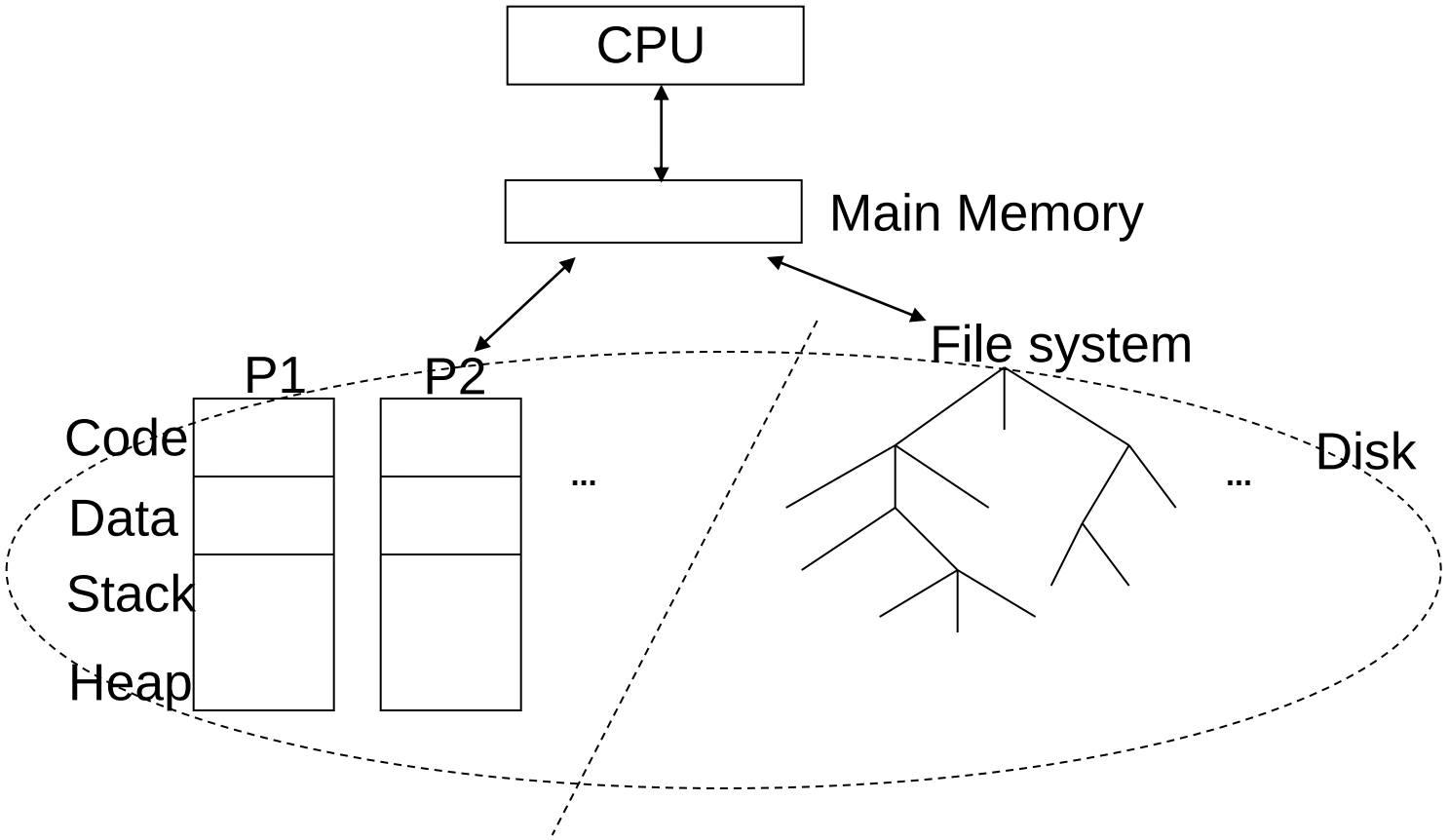
Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks



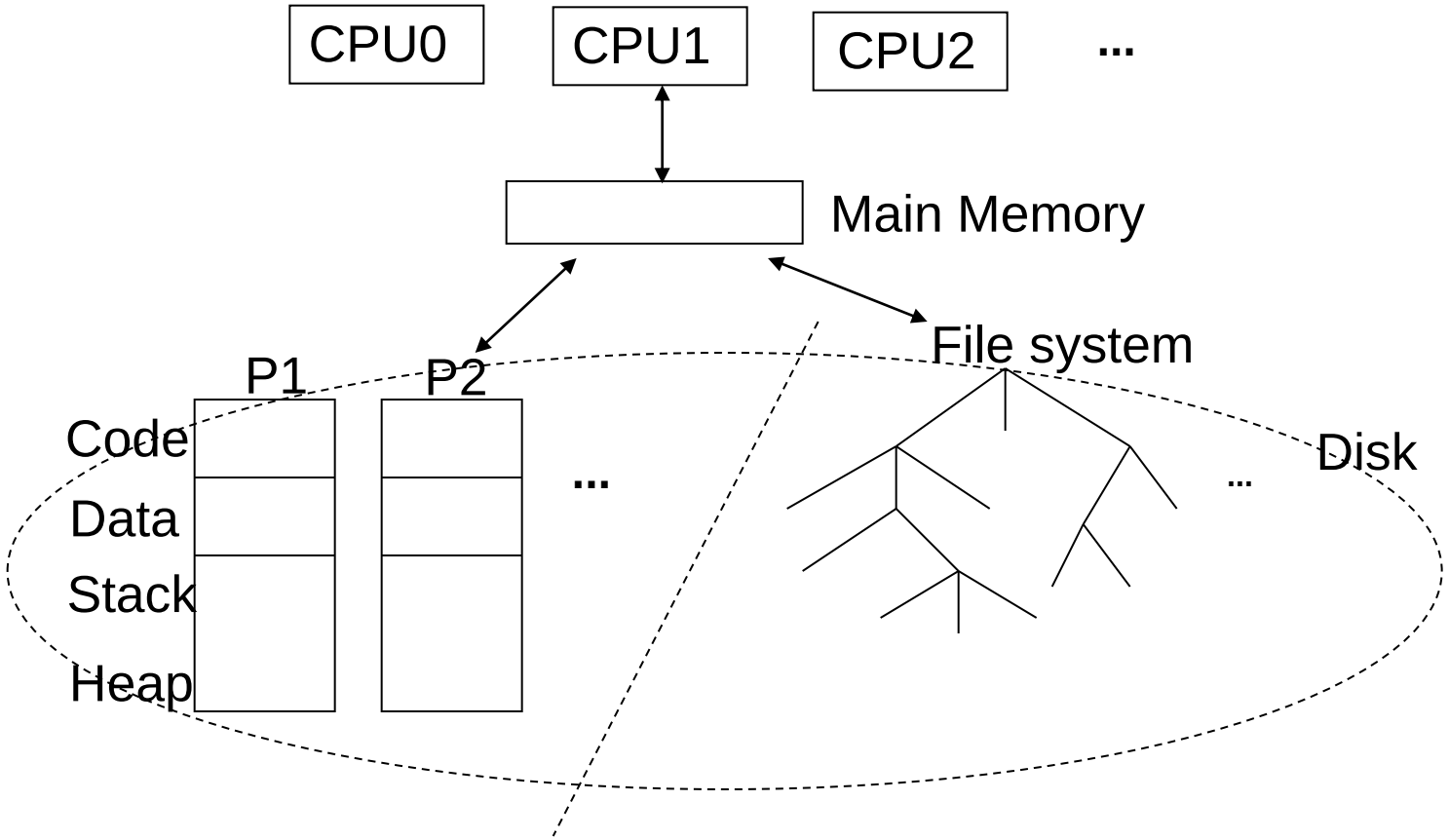
Uniprocessor System

- Multiple virtual addresses and single main memory
  - Execute processes sequentially
  - Virtual memory: protection and sharing



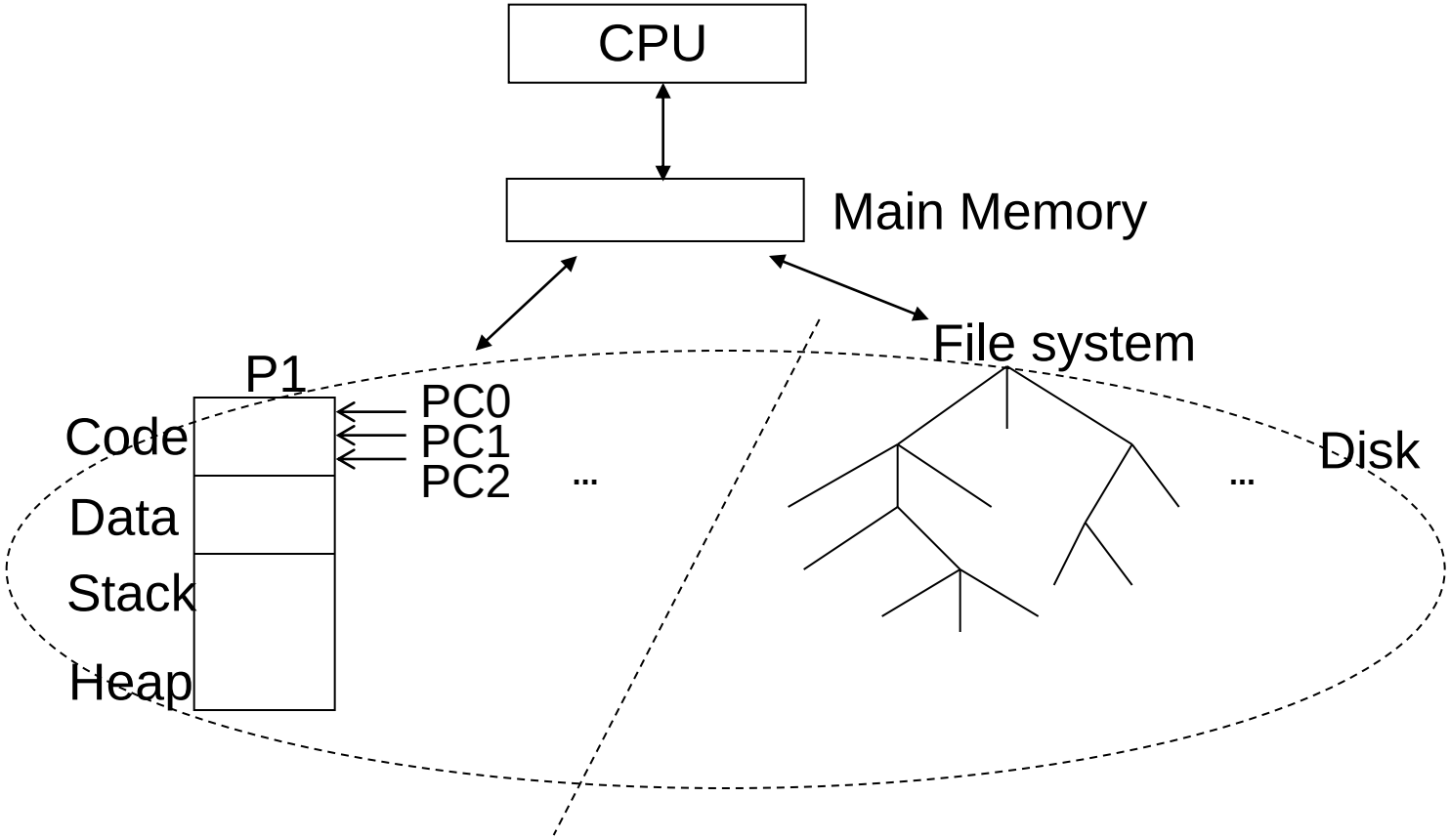
Shared Memory Multiprocessor

- Multiple virtual addresses and single main memory
  - Run processes simultaneously (throughput, resp. time)
  - Virtual memory: protection and sharing



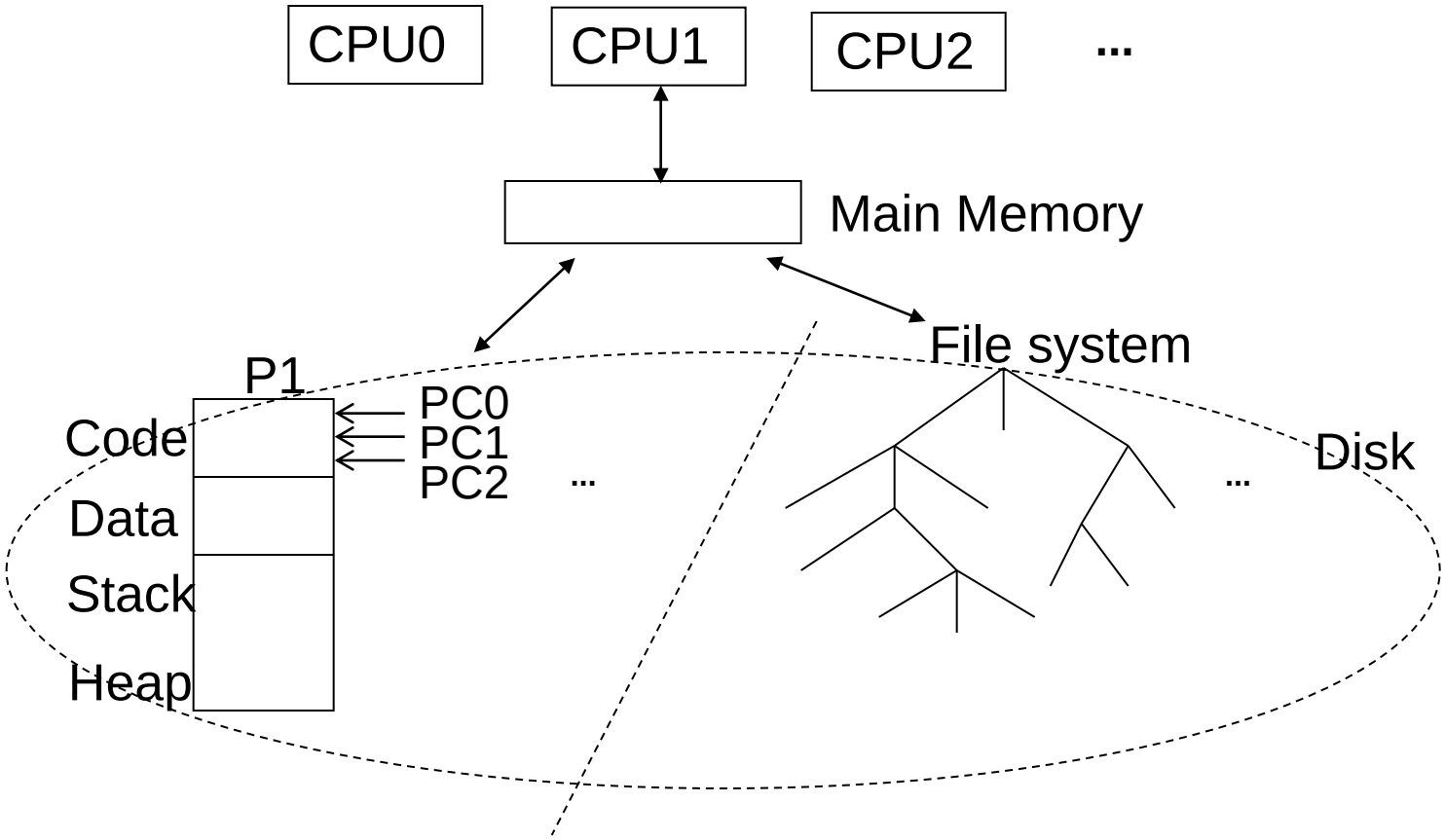
Uniprocessor System

- Single virtual address and single main memory
  - Execute threads sequentially
    - Each has own PC, code area, data area



Shared Memory Multiprocessor

- Single virtual address and single main memory
  - Run threads simultaneously (throughput, resp. time)
    - Each has own PC, code area, data area



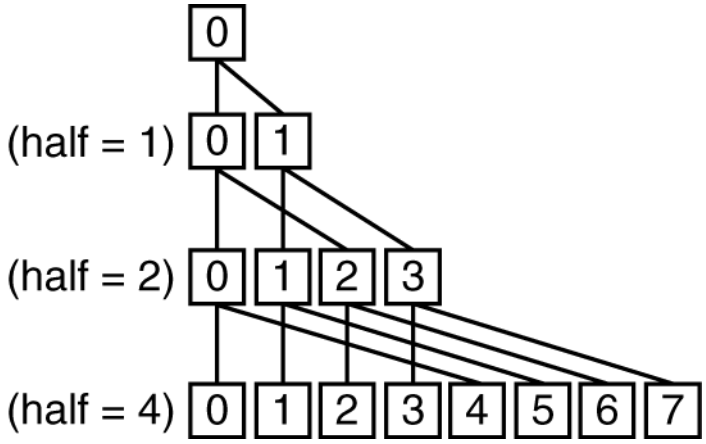
Example: Sum Reduction (skip)

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
     i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction (skip)




```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

OpenMP (skip)

- A limited but popular example from hundreds of attempted parallel programming systems
- API, compiler directives, environment variables and runtime libraries that extend standard programming languages
  - Portable, scalable, simple programming model for SMP
  - Primary goal: parallelize loop & perform reduction
- Most C compilers support OpenMP
  - Use OpenMP API with UNIX C compiler

    cc –fopenmp foo.c
  - OpenMP extend C using *pragmas*

```
#define P 100
#pragma omp parallel num_threads(P)
```

OpenMP (skip)

- Parallelize loop (assuming *sum* is initialized to 0)

```
#pragma omp parallel for
   for (Pn = 0; Pn < P; Pn = Pn+1)
      for (i= 1000*Pn;i < 1000*(Pn+1);i = i+1)
         sum[Pn] = sum[Pn] + A[i];
```

- Perform reduction

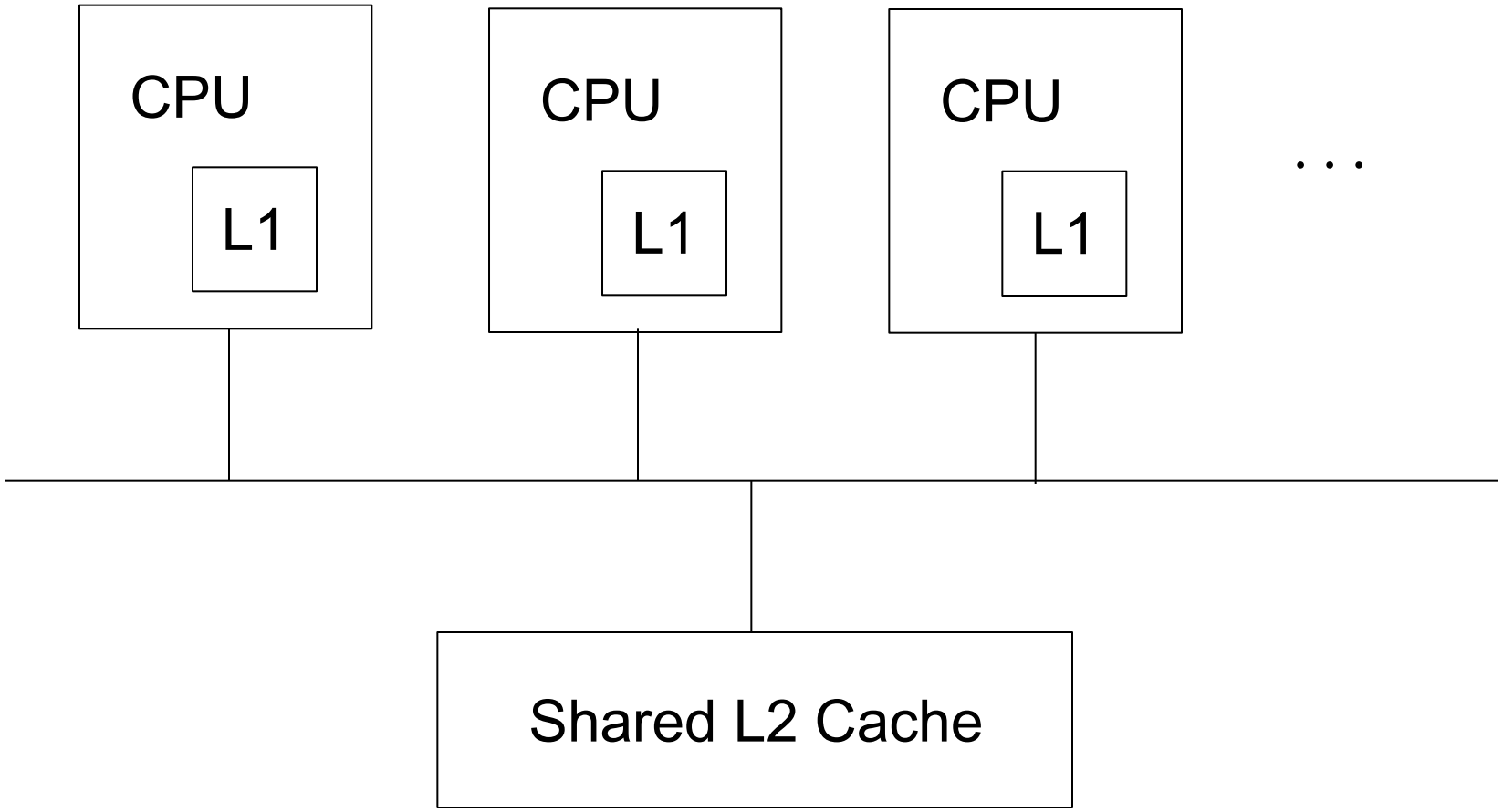
```
#pragma omp parallel for reduction(+ Finalsum)
   for (i = 0; i < P; i += 1)
      Finalsum += sum[i];
```

- Programmers often use more sophisticated parallel programming systems than OpenMP

Section 5.10

Parallelism and Memory Hierarchy: cache Coherence

Cache Coherence Problem



Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Section 6.7

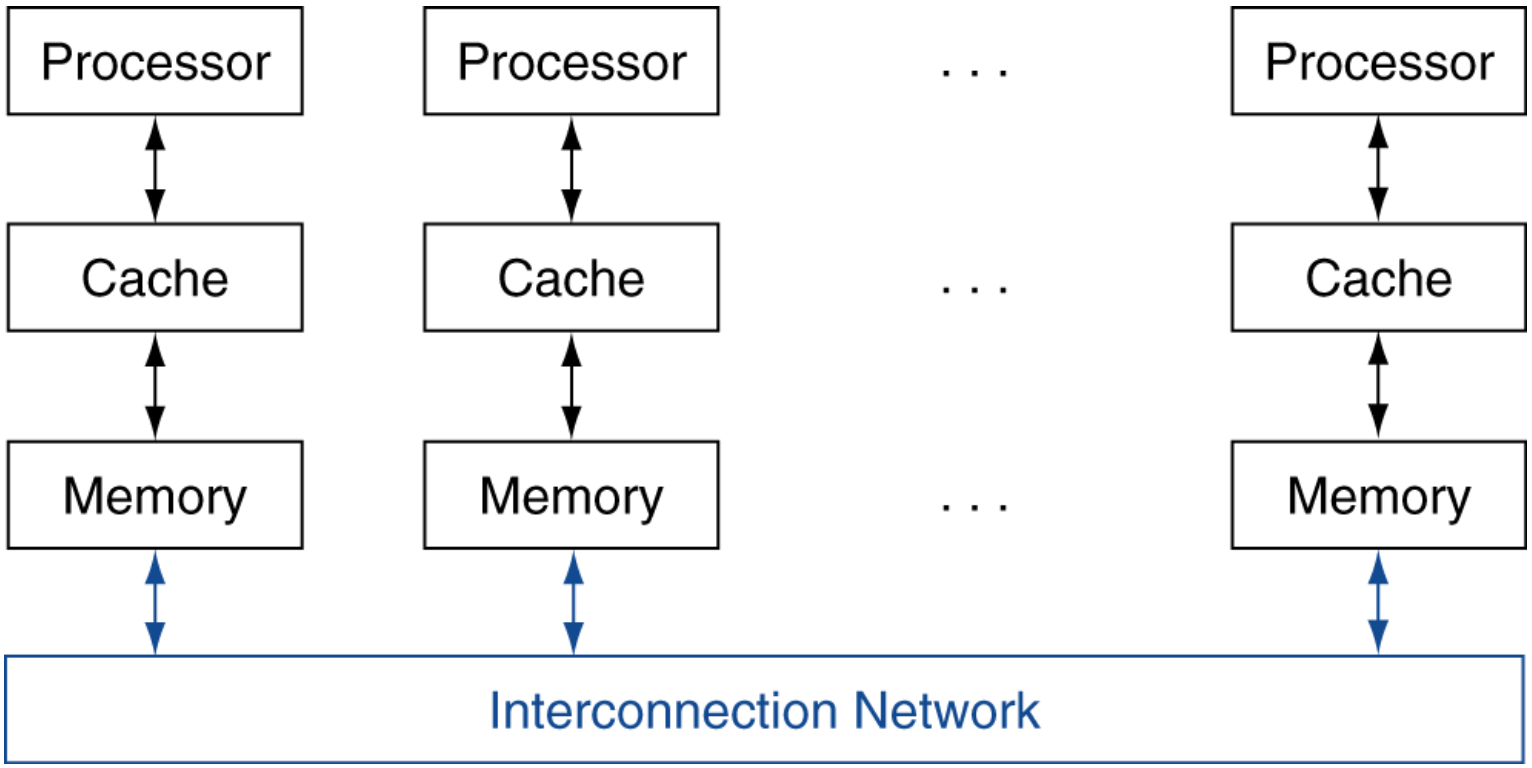
Clusters, warehouse-scale computers, and other message-passing multiprocessors

Message Passing Machines

- Each processor has private physical address space
  - Alternative to sharing an address space
- Hardware sends/receives messages between processors (thus the name)
  - Explicit communication
  - Coordination is built in with message passing
- Interconnection network
  - Custom vs. local area networks like Ethernet
    - Many supercomputers use custom networks
    - Cost of custom network not justifiable for other applications

Message Passing

- Easier to build much larger systems
- Internet services you depend on depend on these large scale systems



Message Passing Machines (vs. SMPs)

- Upside
  - Much easier for hardware designers to build
  - Explicit communication: an advantage to programmers
    - Fewer performance surprises than with implicit cache-coherent shared memory computers
- Downside
  - Harder to port sequential program to message passing computer
  - Every communication must be identified in advance or the program doesn’t work

Sum Reduction (Again) (skip)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
   sum = sum + AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, …

Sum Reduction (Again) (skip)

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition

Message Passing vs. Shared Memory

- Which is right path to high-performance?
- No confusion in the marketplace
  - Multicore processors use shared physical memory
  - Nodes of a cluster communicate with each other with message passing
- What is a cluster?

Section 6.6

Introduction to Graphics Processing Units

GPUs

- GPU: an exotic type of MIMD architecture
  - Different heritage and thus very different perspective on parallel programming challenge
  - Share address space
- Classic MIMD architectures
  - SMP
  - Clusters