CMPE 411
Computer Architecture

Lecture 28

Introduction to Multiprocessor Systems

December 7, 2017

www.csee.umbc.edu/~younis/CMPE411/ CMPE411.htm

Lecture's Overview

❑ *Previous Lecture*

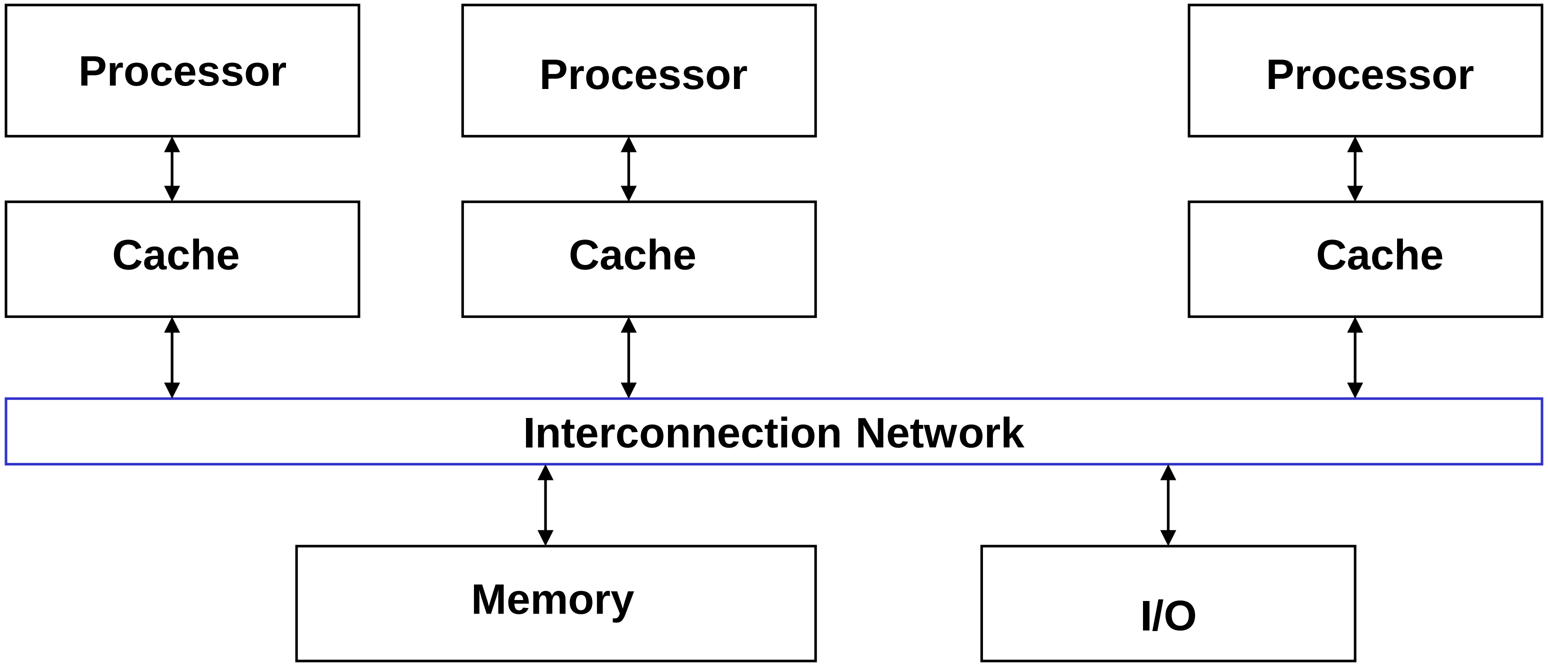
- ➔ Interfacing I/O devices
  - Interfacing with I/O devices
  - Communication with I/O devices
  - Operating System's role
- ➔ Memory to processor interconnect
  - Definition of bus structure
  - Bus transactions
  - Types of buses
  - Bus Standards
- ➔ Bus Performance and Protocol
  - Synchronous versus Asynchronous buses
  - Bandwidth optimization factors
  - Single versus multiple master bus
  - Bus arbitration approaches

❑ *This Lecture*

- ➔ Introduction to multiprocessor systems
- ➔ Issues of parallelism
- ➔ Architecture models for multiprocessors

The Big Picture: Where are We Now?

- Multiprocessor – a computer system with at least two processors



➔ Can deliver high throughput for independent jobs via job-level parallelism or process-level parallelism

➔ And improve the run time of a *single* program that has been specially crafted to run on a multiprocessor - a parallel processing program

* Slide is a courtesy of Mary Jane Irwin

Popular Multiprocessor Systems

Multicores Now Common

- The power challenge has forced a change in the design of microprocessors
  - ➔ Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core – Chip Multicore microProcessors (CMPs) – in a single IC
  - ➔ The number of cores is expected to double every two years

Product	AMD Barcelona	Intel Nehalem	IBM Power 6	Sun Niagara 2
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Power	120 W	~100 W?	~100 W?	94 W

Computing Cluster

- A cluster – a set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
  - ➔ Search engines, Web servers, email servers, databases, …
- A key challenge is to craft parallel (concurrent) programs that have high performance on multiprocessors as the number of processors increase:
  - ➔ Load balancing, time for synchronization, overhead for communication

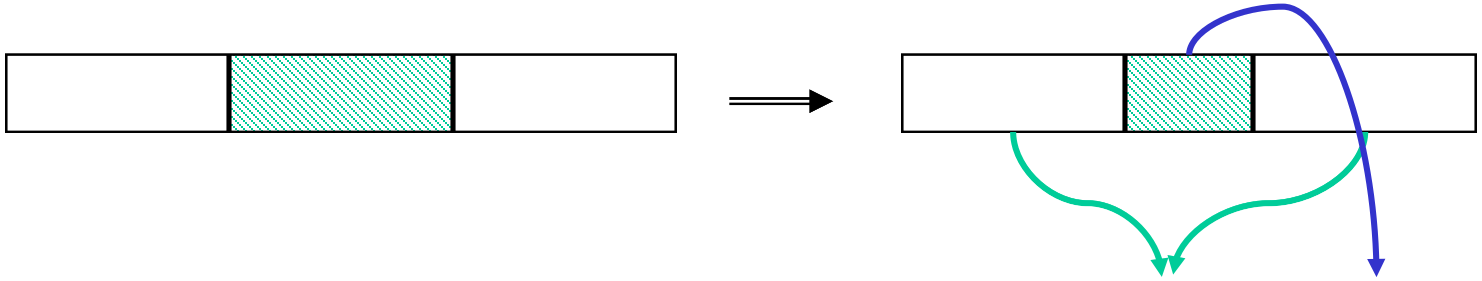
* Slide is a courtesy of Mary Jane Irwin

Encountering Amdahl's Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



ExTime w/ E = ExTime w/o E × ((1-F) + F/S)

Speedup w/ E = 1 / ((1-F) + F/S)

* Slide is a courtesy of Mary Jane Irwin

Example 1: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

  Speedup w/ E = 1/(.75 + .25/20) = 1.31

- What if its usable only 15% of the time?

  Speedup w/ E = 1/(.85 + .15/20) = 1.17

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

  Speedup w/ E = 1/(.001 + .999/100) = 90.99

* Slide is a courtesy of Mary Jane Irwin

Opportunities for Applications

Scientific Computing

- Nearly Unlimited Demand (Grand Challenge):

App	*Perf (GFLOPS)*	*Memory (GB)*
48 hour weather	0.1	0.1
72 hour weather	3	1
Pharmaceutical design	100	10
Global Change, Genome	1000	1000

- Successes in some real industries:
  - ➔Petrolium: reservoir modeling
  - ➔Automotive: crash simulation, drag analysis, engine
  - ➔Aeronautics: airflow analysis, engine, structural mechanics
  - ➔Pharmaceuticals: molecular modeling
  - ➔Entertainment: full length movies ("Toy Story")

Commercial Applications

- Transaction processing, File servers, electronic CAD simulation (multiple processes), WWW search engines
- Examples: IBM RS6000, Tandem (Compaq) Himilaya

* Slide is a courtesy of Dave Patterson

Parallel Computers

- Definition: “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

  Almasi and Gottlieb, *Highly Parallel Computing* ,1989

- Parallel machines is expected to have a bigger role in the future since:
  - Microprocessors are likely to remain dominant in the uniprocessor arena and the logical way to extend the performance is by connecting multiple microprocessors
  - It is not expected that the microprocessor technology will keep the pace of performance improvement given the increased level of complexity
  - There has been steady progress in software development for parallel architectures in recent years
- Questions about parallel computers:
  - ➔How large a collection?
  - ➔How powerful are processing elements?
  - ➔How do they cooperate and communicate?
  - ➔How are data transmitted?
  - ➔What type of interconnection?
  - ➔What are HW and SW primitives for programmer?
  - ➔Does it translate into performance?

* Slide is a courtesy of Dave Patterson

Taxonomy of Parallel Architecture

Flynn Categories

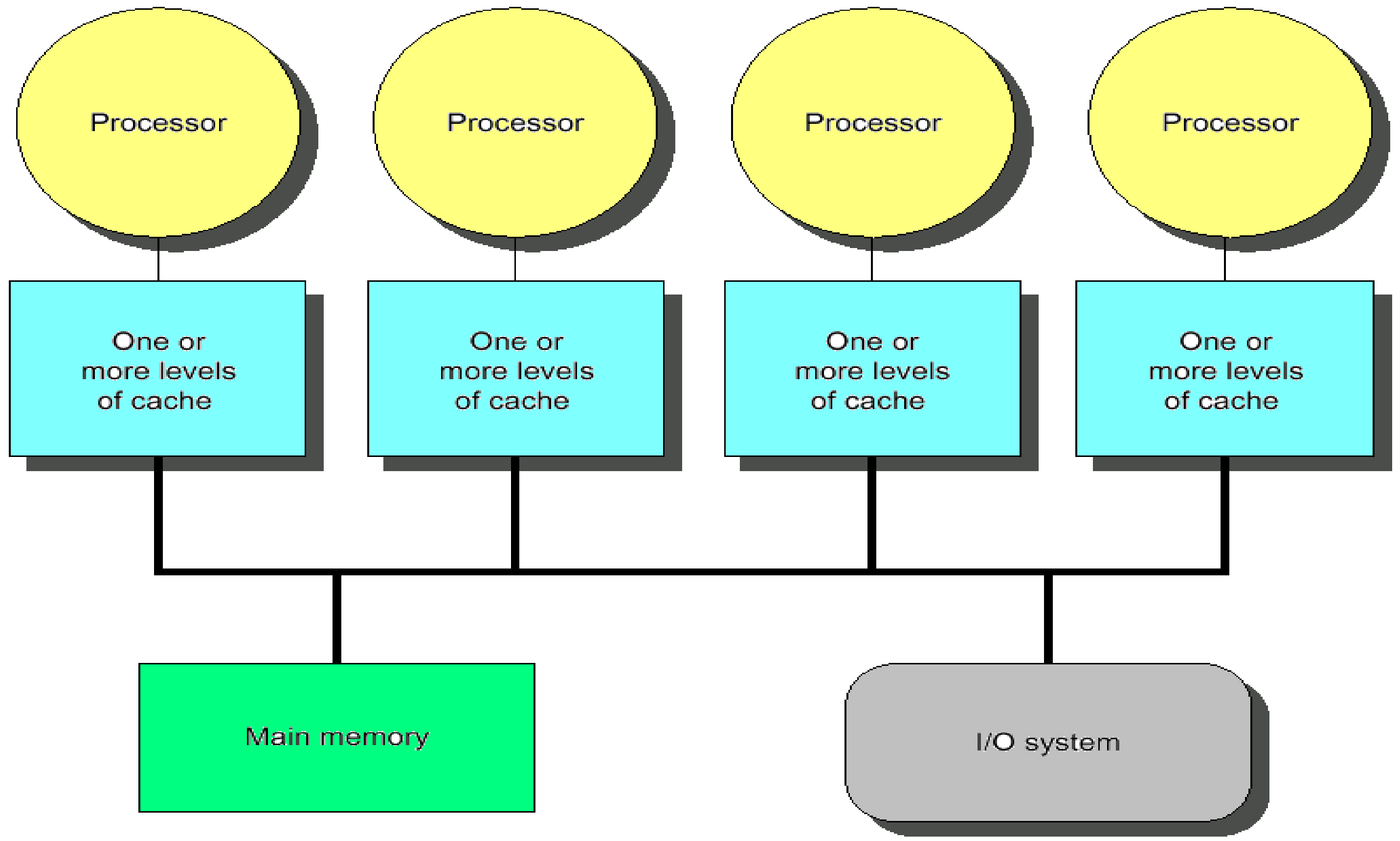
- ❑ SISD (Single Instruction Single Data): Uniprocessors
- ❑ MISD (Multiple Instruction Single Data): No commercial machine yet
- ❑ SIMD (Single Instruction Multiple Data): Vector machine
  - ➔Examples: Illiac-IV, CM-2
    - Simple programming model, Low overhead, Flexibility
- ❑ MIMD (Multiple Instruction Multiple Data)
  - ➔Two classes: centralized shared memory and distributed memory architectures
  - ➔Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - Flexible, *Use off-the-shelf micros*

Level of Parallelism

- ❑ Bit level parallelism: 1970 to 1985
  - ➔4 bits, 8 bit, 16 bit, 32 bit microprocessors
- ❑ Instruction level parallelism (ILP): 1985 through today
  - ➔Pipelining, Superscalar, VLIW, Out-of-Order execution
- ❑ Process/Thread level parallelism: mainstream for general purpose computing
  - ➔Servers are parallel
  - ➔High-end Desktop multi (dual, quad, …) processor PC

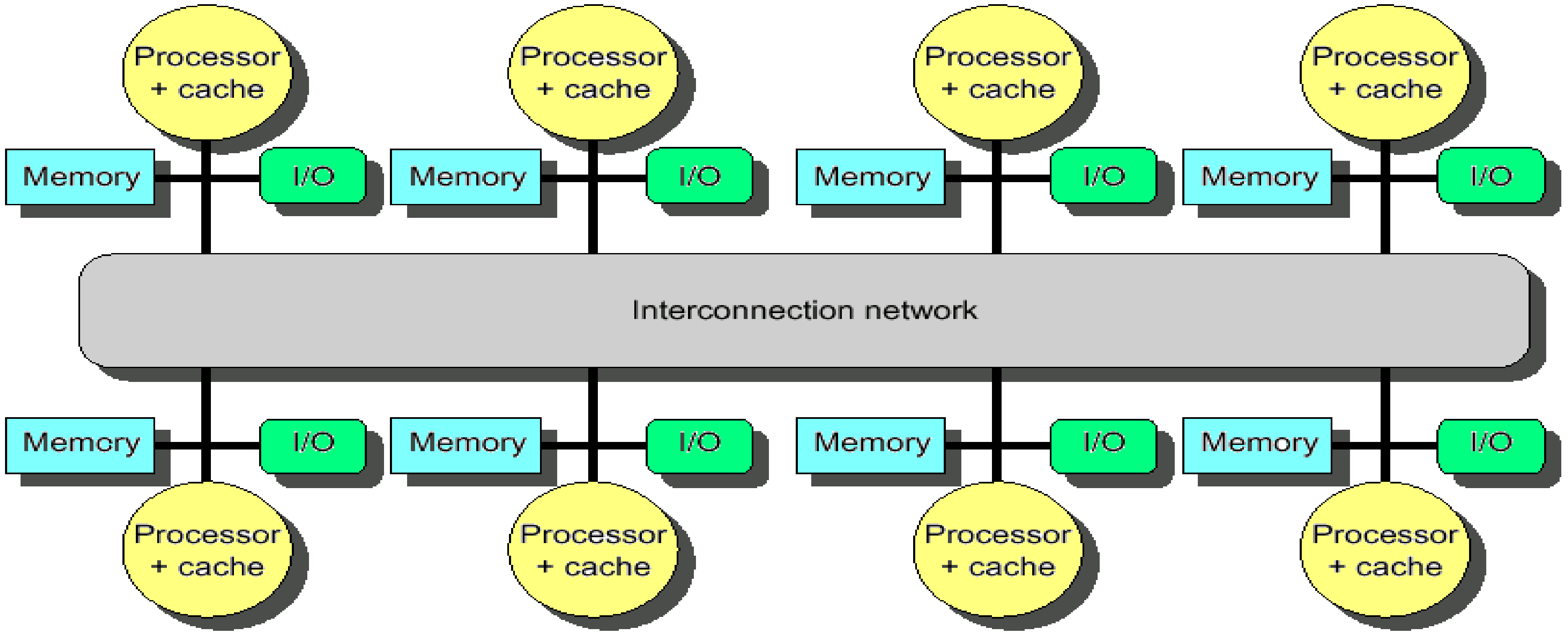
* Slide is a courtesy of Dave Patterson

Centralized Shared Memory MIMD


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

- Processors share a single centralized memory through a bus interconnect
- Feasible for small processor count to limit memory contention
- Usually called “Uniform Memory Access” or UMA model
- Centralized shared memory architectures are the most common form of MIMD design

Distributed Memory MIMD



- ❑ Uses physically distributed memory to support large processor counts (to avoid memory contention)
- ❑ Processor nodes can have some I/O devices and might be composed of multiple processors in a centralized shared memory architecture (clustering)
- ❑ Distributed memory MIMD provides the following advantages
  - ➢Allows cost-effective way to scale the memory bandwidth
  - ➢Reduces memory latency
- ❑ Increased complexity of communicating data is the major disadvantage

Communication Models

- Shared Memory
  - ➔ Processors communicate with shared address space
  - ➔ Easy on small-scale machines
  - ➔ Properties:
    - Model of choice for uniprocessors, small-scale multiprocessor
    - Ease of programming
    - Lower latency
    - Easier to use hardware controlled caching
    - Difficult to handle node failure
- Message passing
  - ➔ Processors have private memories, communicate via messages
  - ➔ Properties:
    - Less hardware, easier to design
    - Focuses attention on costly non-local operations
- Can support either SW models on any HW bases

* Slide is a courtesy of Dave Patterson

Summing 100,000 Numbers on 100 Proc.

- Processors start by running a loop that sums their subset of vector A numbers (vectors A and `sum` are shared variables, Pn is the processor's number, `i` is a private variable)

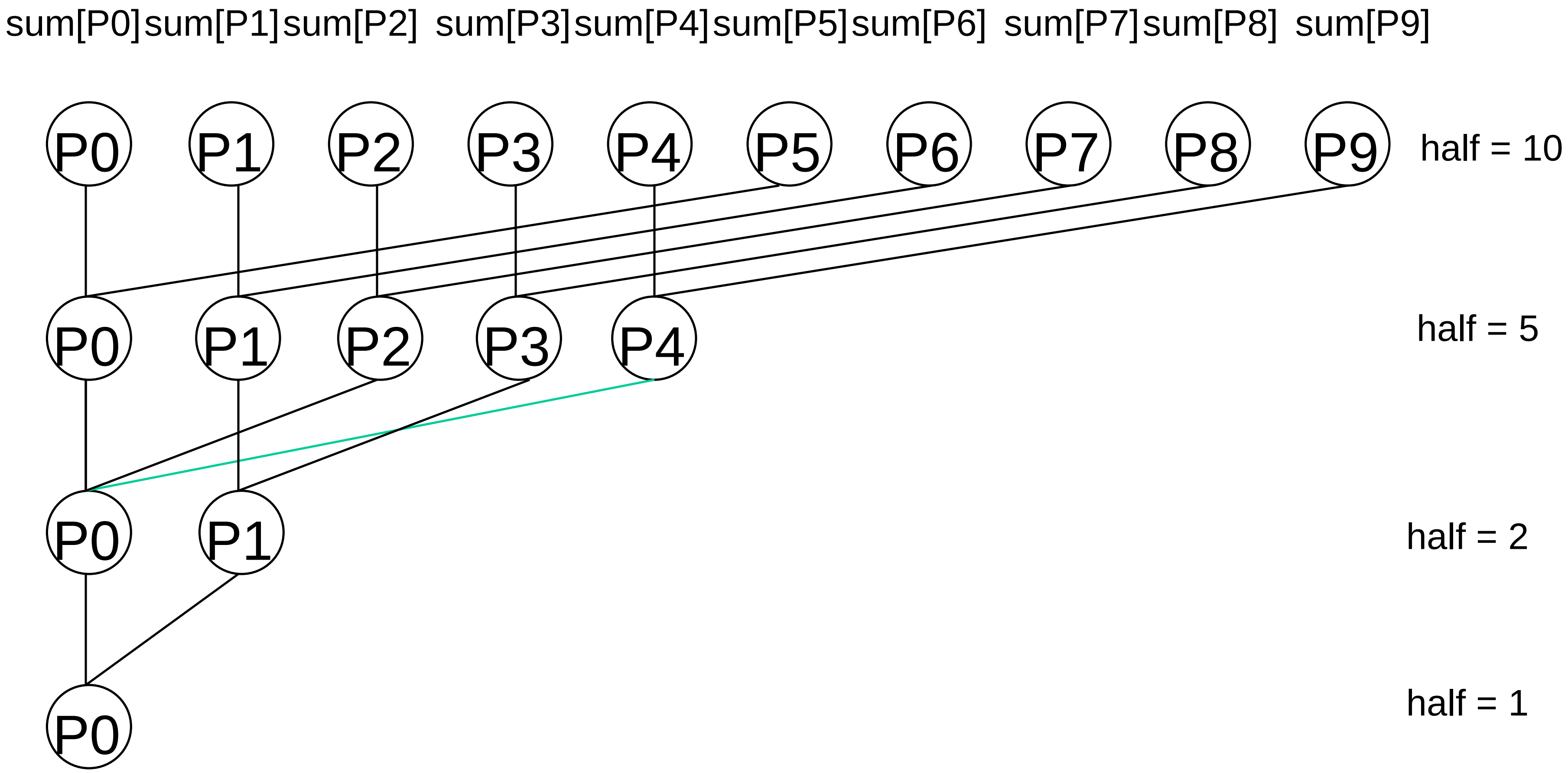
```
sum[Pn] = 0;
for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- The processors then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of processors)) – reduction

```
repeat
  synch();                  /*synchronize first
  if (half%2 != 0 && Pn == 0)
      sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half]
until (half == 1);          /*final sum in sum[0]
```

* Slide is a courtesy of Mary Jane Irwin

An Example with 10 Processors
(Shared Memory Model)


sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
half = 10
half = 5
half = 2
half = 1

* Slide is a courtesy of Mary Jane Irwin

Summing 100,000 Numbers on 100 Proc. (Message Passing Model)

- Start by distributing 1000 elements of vector A to each of the local memories and summing each subset in parallel

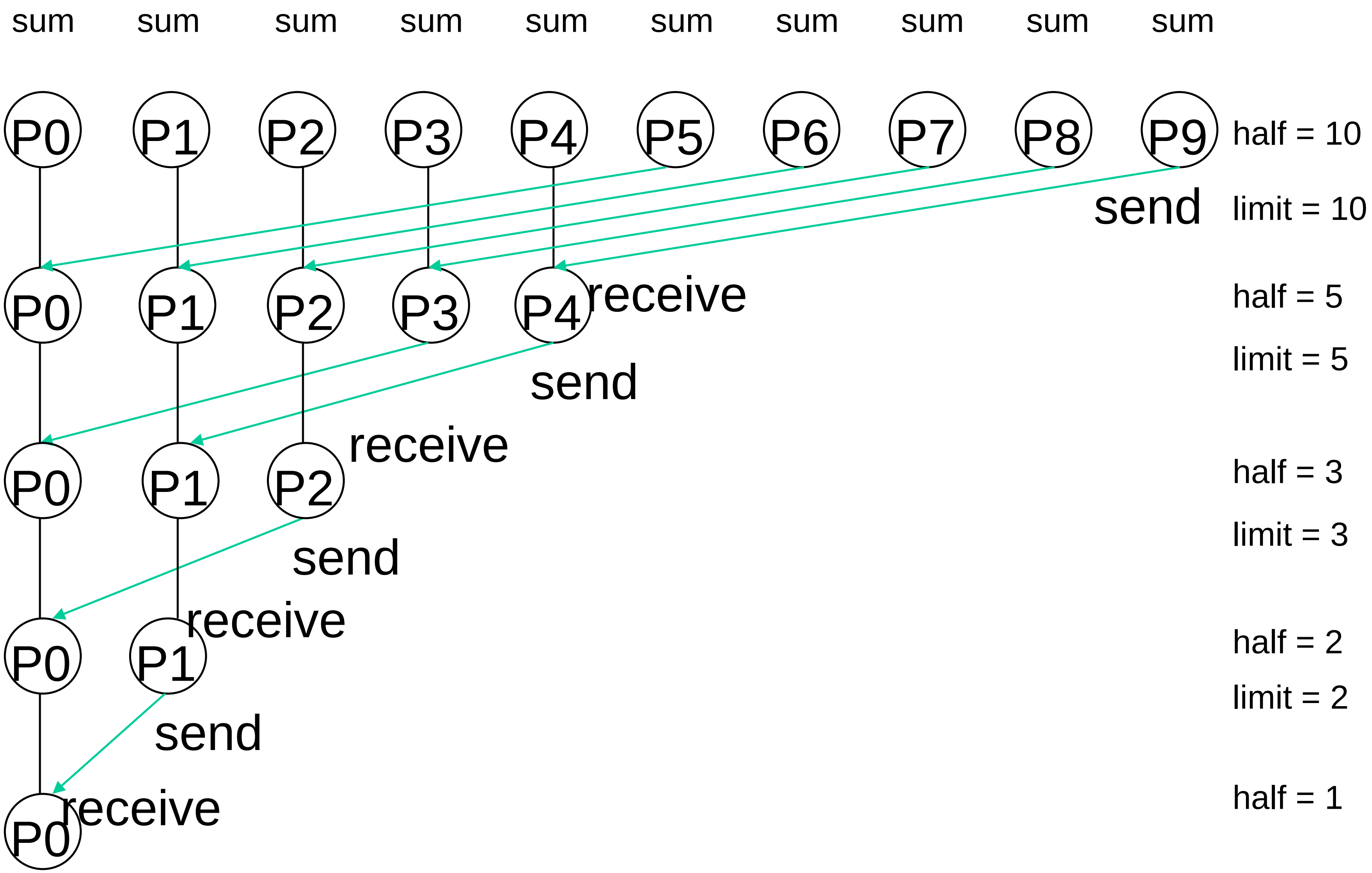
```
sum = 0;
for (i = 0; i<1000; i = i + 1)
   sum = sum + Al[i];     /* sum local array subset
```

- The processors then coordinate in adding together the sub sums (Pn is the number of processors, `send(x,y)` sends value y to processor x, and `receive()` receives a value)

```
half = 100;
limit = 100;
repeat
  half = (half+1)/2;     /*dividing line
  if (Pn>= half && Pn<limit) send(Pn-half,sum);
  if (Pn<(limit/2)) sum = sum + receive();
  limit = half;
until (half == 1);       /*final sum in P0’s sum
```

* Slide is a courtesy of Mary Jane Irwin

An Example with 10 Processors
(Message Passing Model)



* Slide is a courtesy of Mary Jane Irwin

Pros and Cons of Message Passing

- Message sending and receiving is *much* slower than addition, for example
- But message passing multiprocessors and much easier for hardware designers to design
  - ➔Don’t have to worry about cache coherency for example
- The advantage for programmers is that communication is explicit, so there are fewer “performance surprises” than with the implicit communication in cache-coherent SMPs.
  - ➔Message passing standard MPI-2 (www.mpi-forum.org)
- However, it is harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
  - ➔With cache-coherent shared memory the hardware figures out what data needs to be communicated

* Slide is a courtesy of Mary Jane Irwin

Fundamental Issues

❶ ***Naming***: how to solve large problem fast

- ➔what data is shared
- ➔how it is addressed
- ➔what operations can access data
- ➔how processes refer to each other

- ❑ Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for message passing
- ❑ Choice of naming affects replication of data; via load in cache memory hierarchy or via SW replication and consistency
- ❑ Global physical address space: any processor can generate, address and access it in a single operation
- ❑ Global virtual address space: if the address space of each process can be configured to contain all shared data of the parallel program
  - ➔memory can be anywhere: virtual address translation handles it
- ❑ Segmented shared address space: locations are named <process number, address>
  - ➔uniformly for all processes of the parallel program

* Slide is a courtesy of Dave Patterson

Fundamental Issues

❷ ***Synchronization***: To cooperate, processes must coordinate

- ❑ Message passing is implicit coordination with transmission or arrival of data
- ❑ Shared address => additional operations to explicitly coordinate: e.g., write a flag, awaken a thread, interrupt a processor

❸ ***Latency and Bandwidth***

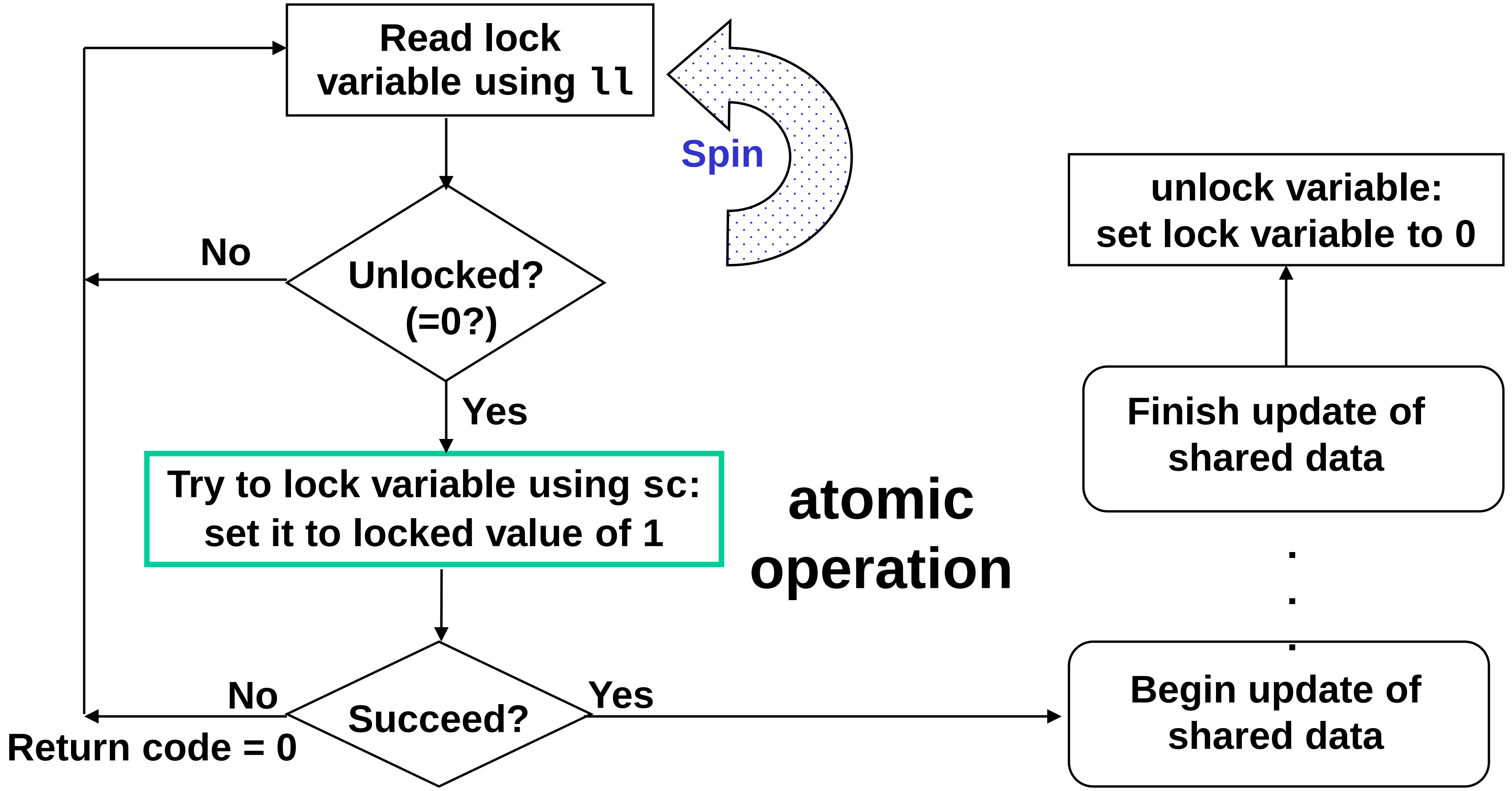
- ❑ Bandwidth
  - ➔Need high bandwidth in communication
  - ➔Cannot scale, but stay close
  - ➔Match limits in network, memory, and processor
  - ➔Overhead to communicate is a problem in many machines
- ❑ Latency
  - ➔Affects performance, since processor may have to wait
  - ➔Affects ease of programming, since requires more thought to overlap communication and computation
- ❑ Latency Hiding
  - ➔How can a mechanism help hide latency?
  - ➔Examples: overlap message send with computation, pre-fetch data, switch to other tasks

* Slide is a courtesy of Dave Patterson

Synchronization

- Why synchronize?
  - ➔ Need to know when it is safe for different processes to use shared data
  - ➔ To preserve execution semantics of multiprocessor applications
- Synchronization mechanisms are typically built with user-level software routines (e.g. *locks* and *barriers*) that rely on hardware-supplied primitives
- The key hardware-supplied synchronization capability is an uninterruptible instruction to atomically retrieve and change a value
- The implementation of hardware synchronization instructions introduces additional level of design complexity
- Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins
- In large-scale machines or high-contention situations, synchronization can be a performance bottleneck due to the latency introduced by contention
- Issues for synchronization:
  - ➔ Negative impact on the execution time of the multiprocessor application
  - ➔ Interactions with cache coherence protocols (avoiding deadlocks)
  - ➔ Interface with user level programs

Spin Lock Synchronization


Read lock variable using ll
Spin
Unlocked? (=0?)
No
Yes
Try to lock variable using sc: set it to locked value of 1
atomic operation
Succeed?
No
Return code = 0
Yes
Begin update of shared data
Finish update of shared data
unlock variable: set lock variable to 0

- Locking can be done via an atomic swap operation. MIPS has load-link (ll) and store-conditional (sc) instructions. Using both instructions a processor reads a location *and* sets it to locked state – test-and-set – in the same bus operation.
- The *single* winning processor will succeed in writing a 1 to the lock variable - all others processors will get a return code of 0

* Slide is a courtesy of Mary Jane Irwin

The Problem of Cache Coherency

- A cache coherence problem arises when the cache reflects a view of memory which is different from reality
- Cache coherence is a common issue when handling the I/O subsystem
- For the centralized shared memory architecture two different processors can view two different values for the same memory location

Time	Event	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

- A memory system is coherent if:

❶ A read by a processor P, to a location X follows a write by P to X, with no writes of X by another processor occurring in between, always returns the value written by P

❷ A read by P to location X that follows a write by another processor to X returns the newly written value if the read and write are sufficiently separated

❸ Writes to the same location are serialized: that is two writes to the same location by any two processors are seen in the same order by all processors (*consistency*)

HW Coherency Solutions

- In a coherent multiprocessor the cache provides both migration and replication of shared data
- Data migration allows low latency access to shared data from a local cache
- Replicating data enables simultaneous access to shared data and limits the potential of contention

Snooping Solution (Snoopy Bus)

- ➔Send all requests for data to all processors
- ➔Processors snoop to see if they have a copy and respond accordingly
- ➔Requires broadcast, since caching information is at processors
- ➔Works well with bus (natural broadcast medium)
- ➔Dominates for small scale machines (most of the market)

Directory-Based Schemes

- ➔Keep track of what is being shared in one centralized place
- ➔Distributed memory => distributed directory for scalability (avoids bottlenecks)
- ➔Send point-to-point requests to processors via network
- ➔Scales better than Snooping
- ➔Actually existed before Snooping-based schemes

* Slide is partially a courtesy of Dave Patterson

Conclusion

❑ Summary

➔ Taxonomy of parallel architecture

- Flynn's categorization (SISD, SIMD, MISD, MIMD)
- Level of parallelism (bit level, instruction level, thread level)
- MIMD architectures (centralized shared memory, distributed memory)
- Programming and communication models

➔ Opportunities for applications

- Scientific applications (weather, pharmaceuticals, aerospace, etc.)
- Commercial applications (transaction processing and web servers)

➔ Communication models

- Distributed shared memory model
- Message passing model

➔ Fundamental issues for parallel computing

- Naming of nodes and data items
- Synchronization of application execution on the different nodes
- Managing latency and bandwidth

❑ Next Lecture

➔ Review

Read sections 7.1-7.6 in of the 4th Ed. Of textbook