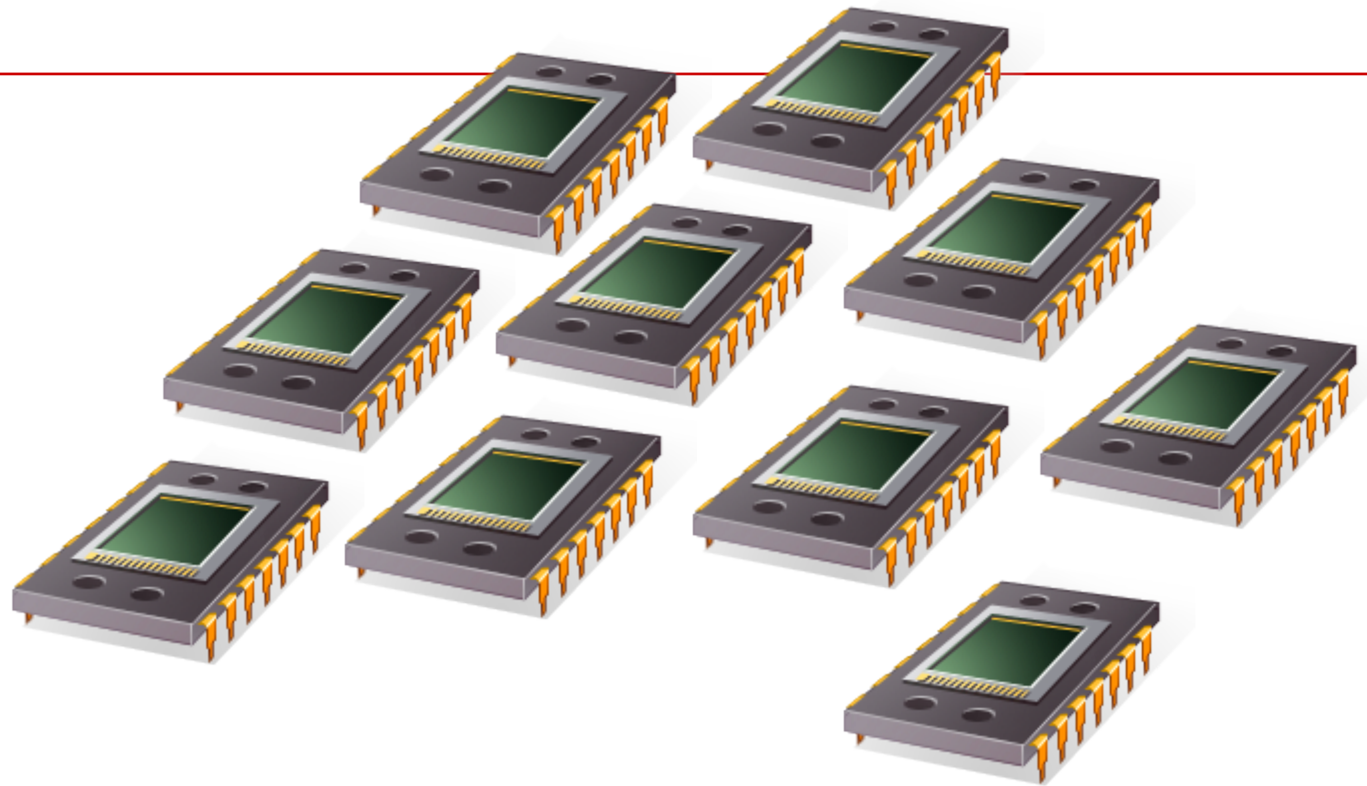


Parallel Architecture, Software And Performance

UCSB CS240A, T. Yang, 2017

Roadmap

- Parallel architectures for high performance computing
- Shared memory architecture with cache coherence
- Performance evaluation
- Parallel program design

Copyright © 2010, Elsevier Inc. All rights Reserved

Flynn's Taxonomy

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

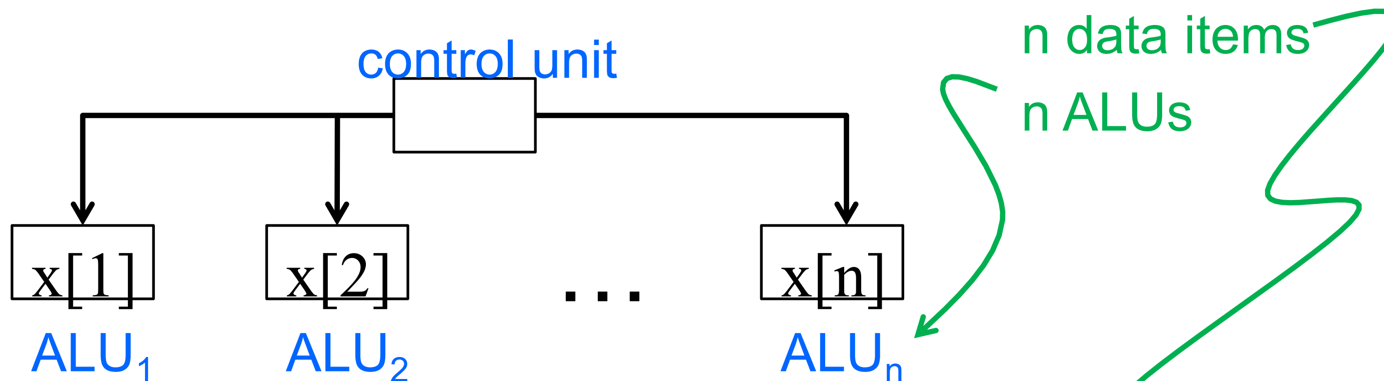
classic von Neumann

not covered

Copyright © 2010, Elsevier
Inc. All rights Reserved

SIMD

- **Parallelism achieved by dividing data among the processors.**
  - Applies the same instruction to multiple data items.
  - Called data parallelism.



Copyright © 2010, Elsevier Inc. All rights Reserved

SIMD drawbacks

- **All ALUs are required to execute the same instruction, or remain idle.**
  - In classic design, they must also operate synchronously.
  - The ALUs have no instruction storage.
- **Efficient for large data parallel problems, but not flexible for more complex parallel problems.**

Copyright © 2010, Elsevier Inc. All rights Reserved

Vector Processors

- **Operate on vectors (arrays) with vector instructions**
  - conventional CPU's operate on individual data elements or scalars.
- **Vectorized and pipelined functional units.**
  - Use vector registers to store data
  - Example:
    - A[1:10]=B[1:10] + C[1:10]
    - Instruction execution
      - Read instruction and decode it
      - Fetch these 10 A numbers and 10 B numbers
      - Add them and save results.

Copyright © 2010, Elsevier Inc. All rights Reserved

Vector processors – Pros/Cons

- **Pros**
  - Fast. Easy to use.
  - Vectorizing compilers are good at identifying code to exploit.
    - Compilers also can provide information about code that cannot be vectorized.
    - Helps the programmer re-evaluate code.
  - High memory bandwidth. Use every item in a cache line.

- **Cons**
  - Don’t handle irregular data structures well
  - Limited ability to handle larger problems (scalability)

Copyright © 2010, Elsevier Inc. All rights Reserved

Graphics Processing Units (GPU)

- Computation for graphic applications is often parallel, since they can be applied to multiple elements in the graphics stream.
- GPU’s can often optimize performance by using SIMD parallelism.
  - The current generation of GPU’s use SIMD parallelism.
  - Although they are not pure SIMD systems.
- **Key Market Players: Intel, NVIDA, AMD**

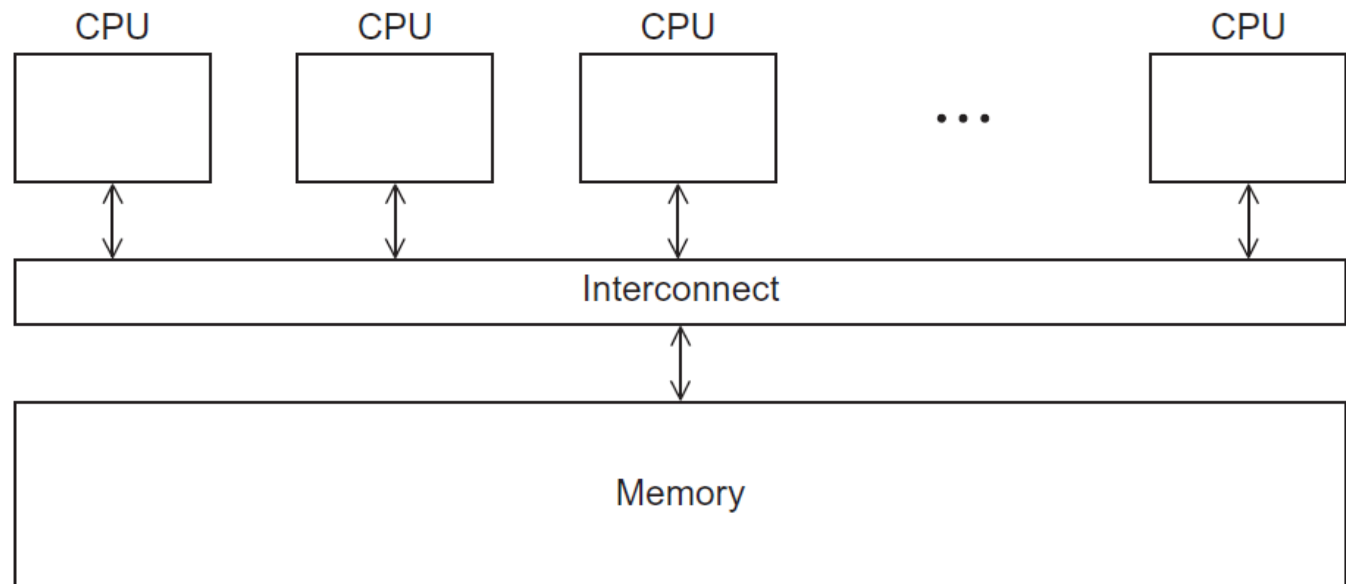
MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.
- Types of MIMD systems
  - Shared-memory systems
    - Most popular ones use multicore processors.
      - (multiple CPU's or cores on a single chip)
  - Distributed-memory systems
    - Computer clusters are the most popular

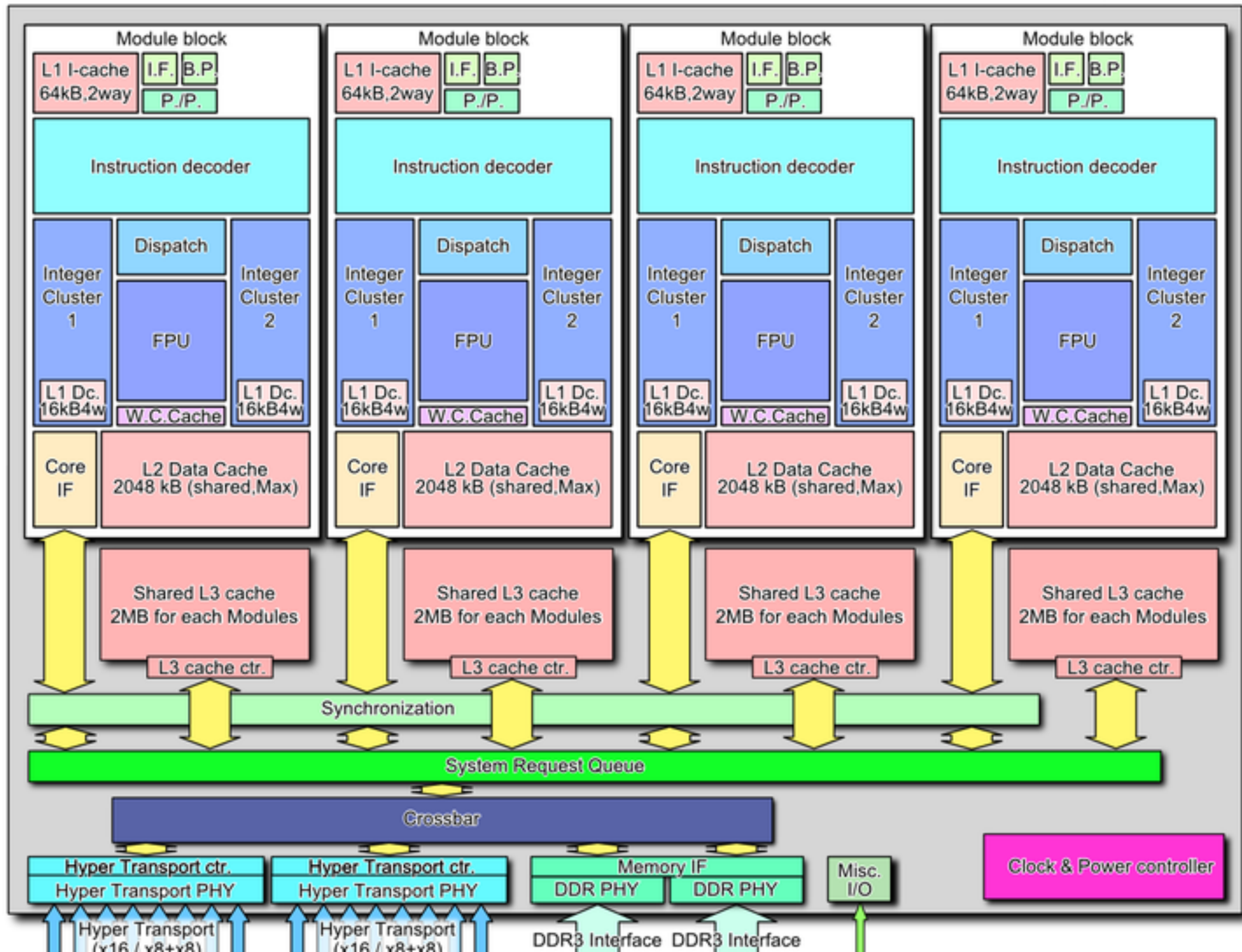
Copyright © 2010, Elsevier Inc. All rights Reserved

Shared Memory System

- **Each processor can access each memory location.**
  - The processors usually communicate implicitly by accessing shared data structures
  - Two designs: UMA (Uniform Memory Access) and NUMA (Non-uniform Memory Access)

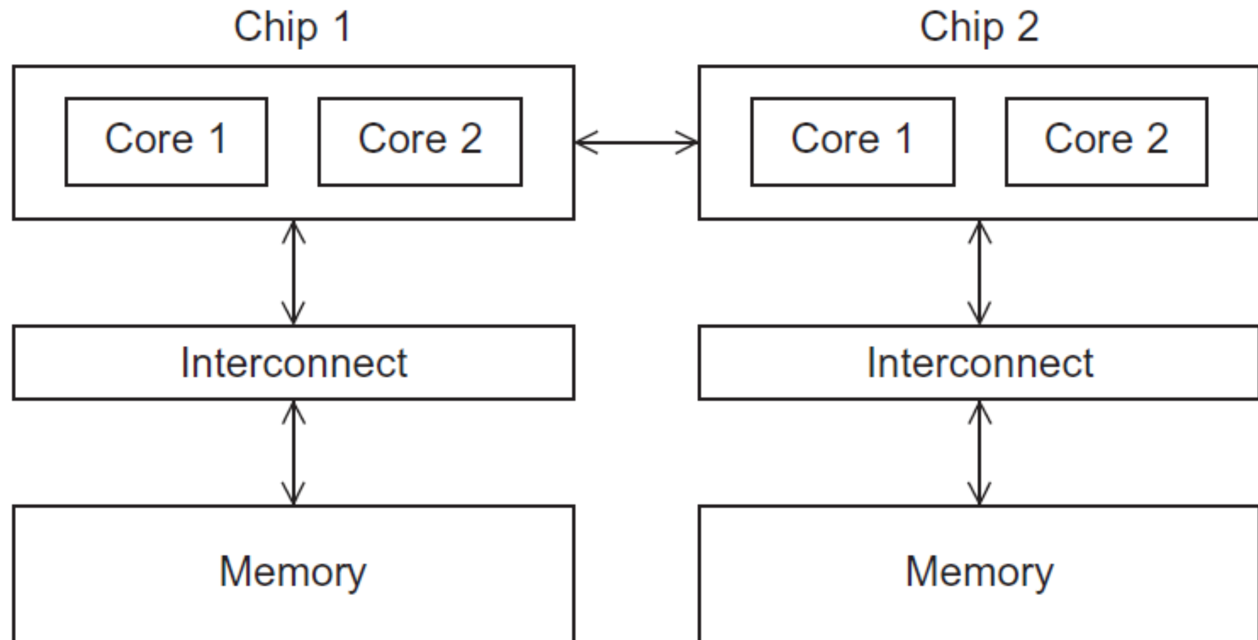


AMD 8-core CPU Bulldozer



sevier
erved

NUMA Multicore System



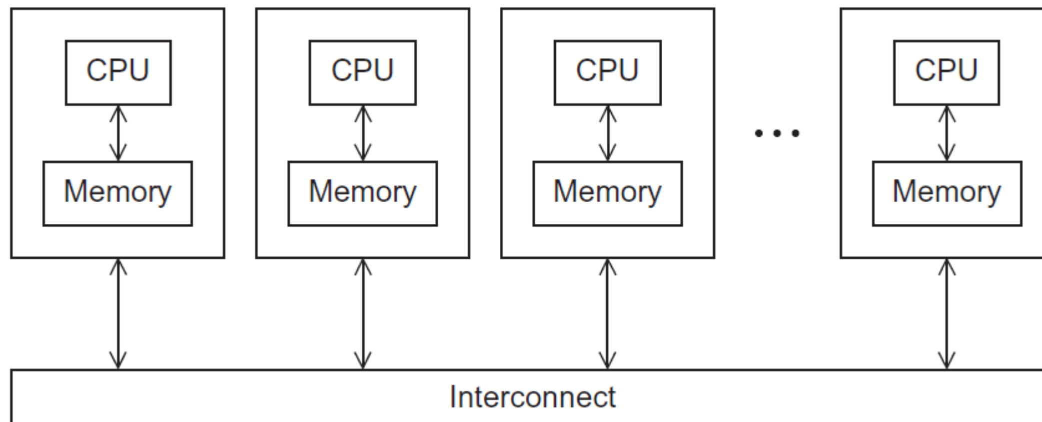
A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Figure 2.6

Copyright © 2010, Elsevier Inc. All rights Reserved

Distributed Memory System

- **Clusters (most popular)**
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.



Copyright © 2010, Elsevier Inc. All rights Reserved

Interconnection networks

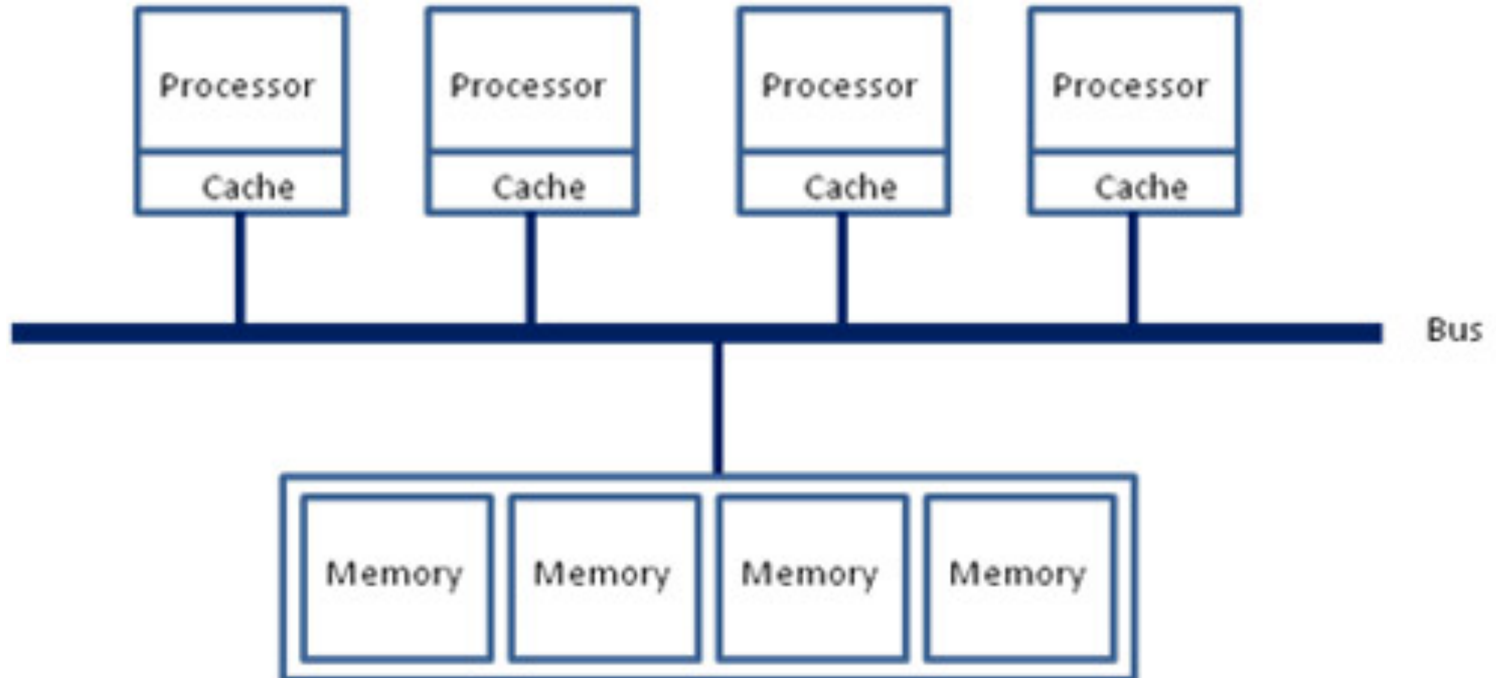
- **Affects performance of both distributed and shared memory systems.**

- **Two categories:**
  - Shared memory interconnects
  - Distributed memory interconnects

Copyright © 2010, Elsevier Inc. All rights Reserved

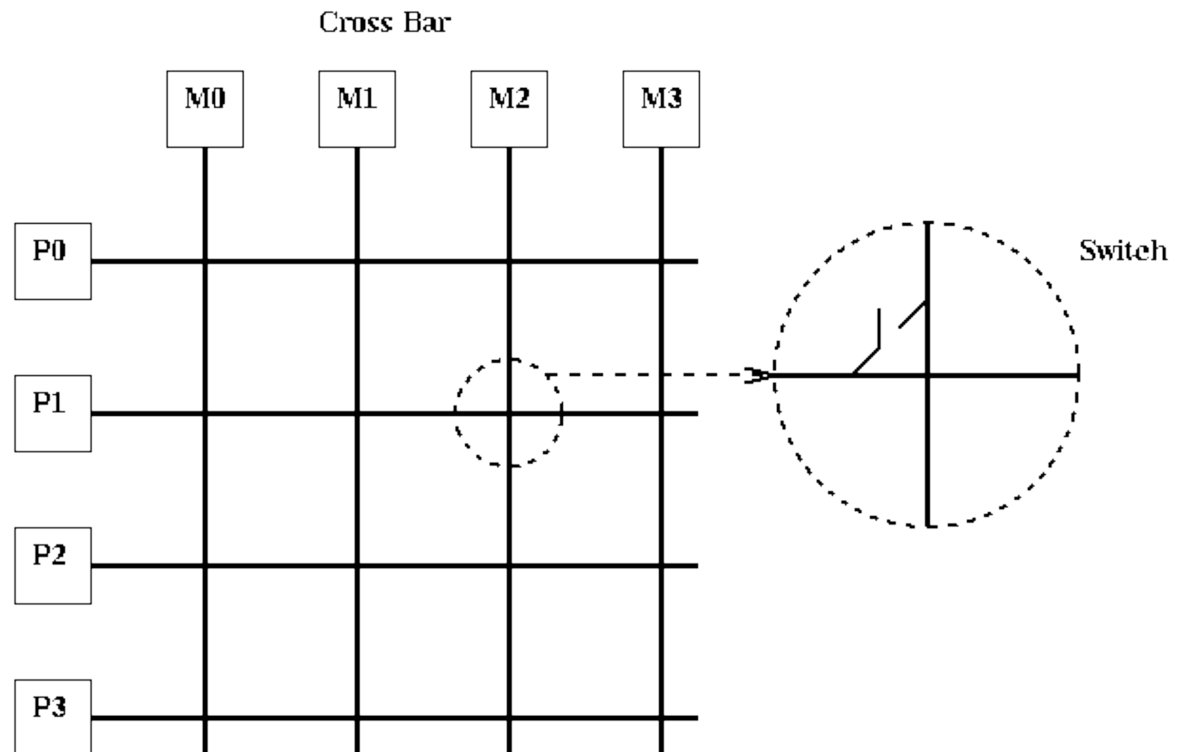
Shared memory interconnects: Bus

- Parallel communication wires together with some hardware that controls access to the bus.
- As the number of devices connected to the bus increases, contention for shared bus use increases, and performance decreases.



Shared memory interconnects: Switched Interconnect

- Uses switches to control the routing of data among the connected devices.
- Crossbar – Allows simultaneous communication among different devices.
  - Faster than buses. But higher cost.

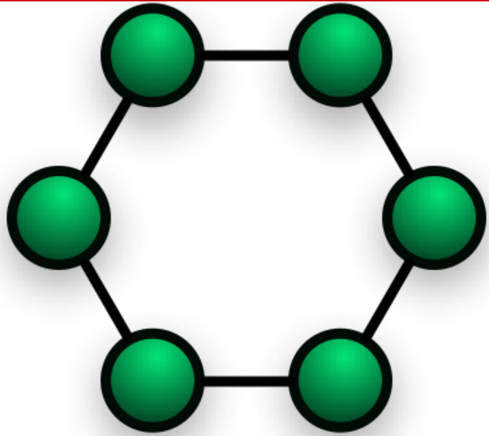


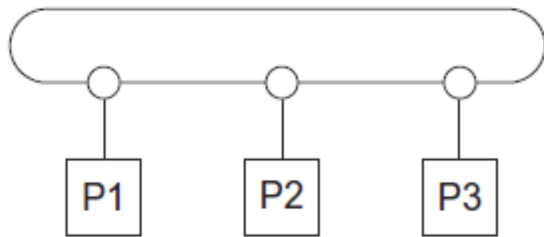
Distributed memory interconnects

- **Two groups**
  - Direct interconnect
    - Each switch is directly connected to a processor memory pair, and the switches are connected to each other.
  - Indirect interconnect
    - Switches may not be directly connected to a processor.

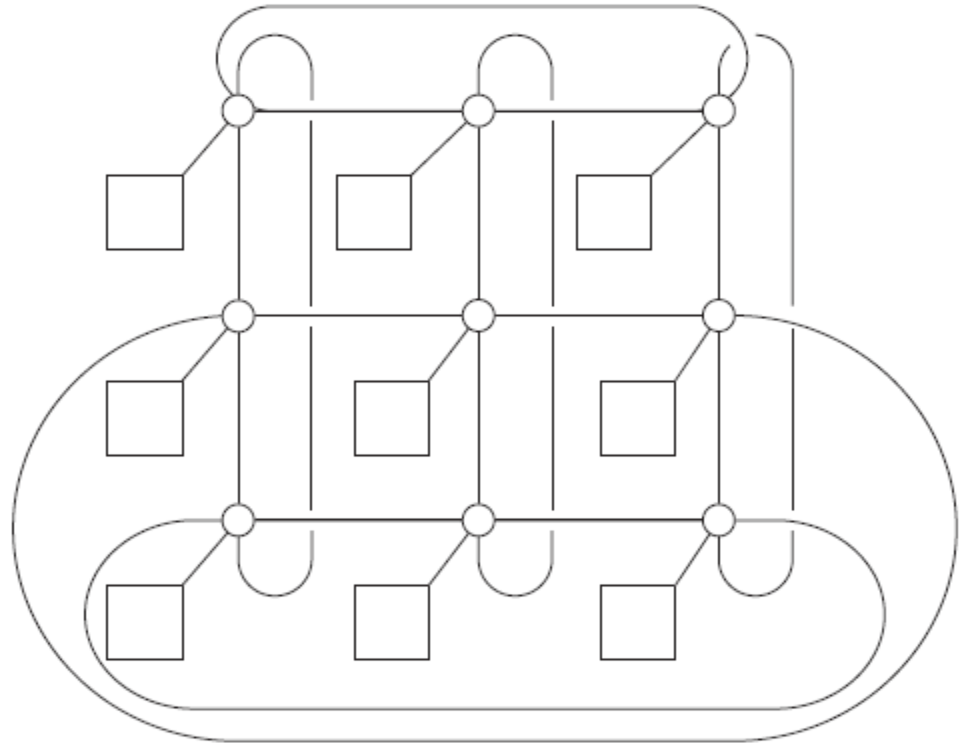
Copyright © 2010, Elsevier
Inc. All rights Reserved

Direct interconnect





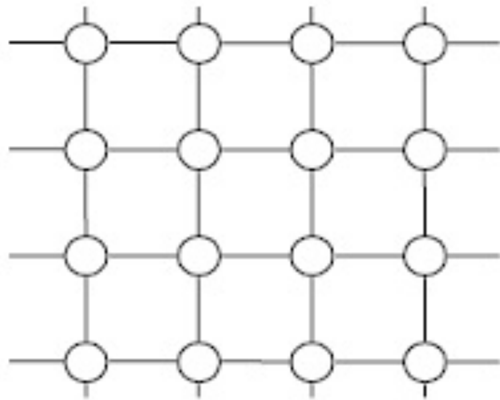
ring



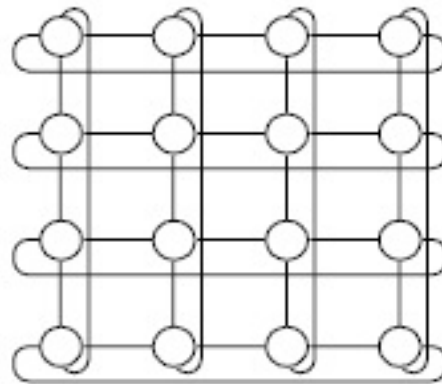
2D torus (toroidal mesh)

Copyright © 2010, Elsevier Inc. All rights Reserved

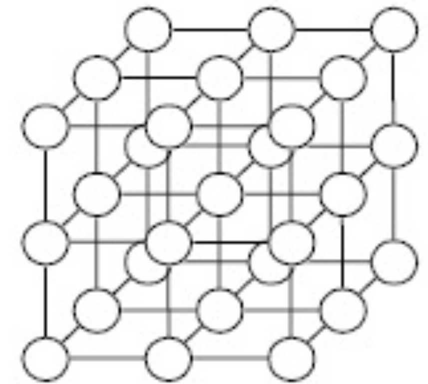
Direct interconnect: 2D Mesh vs 2D Torus



2D mesh



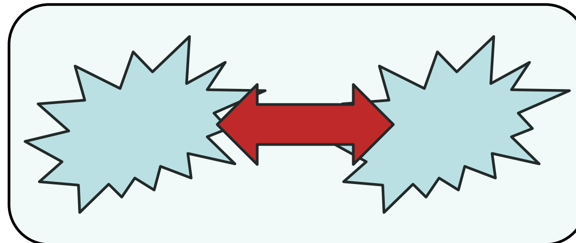
2D torus



3D mesh

How to measure network quality?

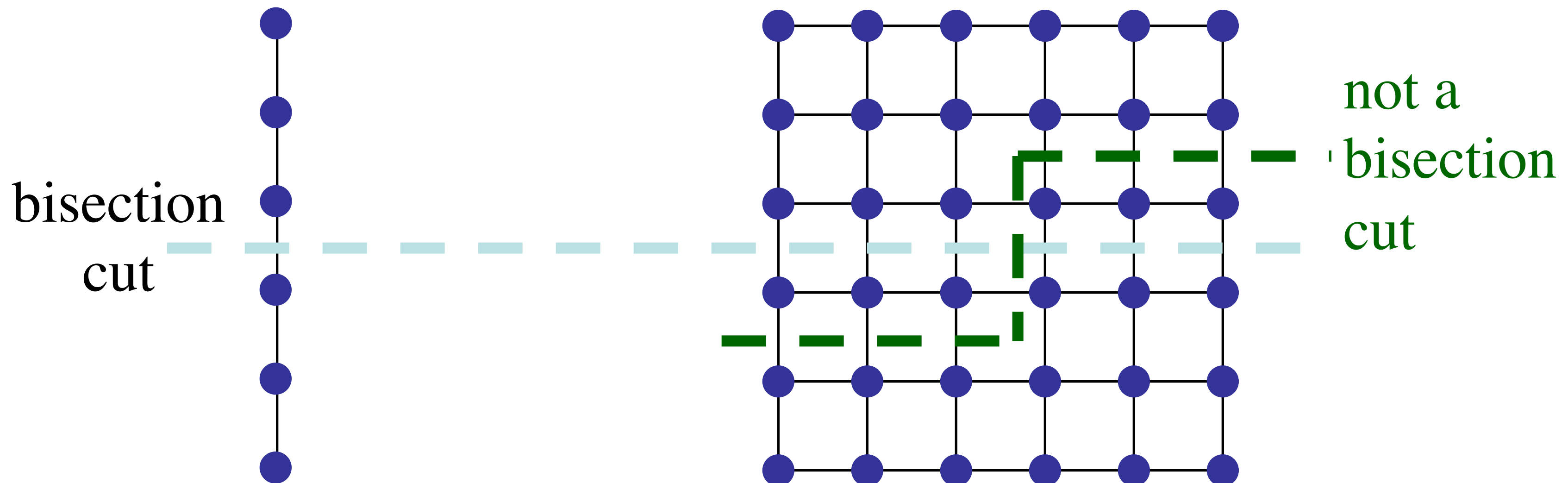
- **Bandwidth**
  - The rate at which a link can transmit data.
  - Usually given in megabits or megabytes per second.
- **Bisection width**
  - A measure of “number of simultaneous communications” between two subnetworks within a network



  - The minimum number of links that must be removed to partition the network into two equal halves
    - 2 for a ring
  - Typically divide a network by a line or plane (bisection cut)

Bisection width vs Bisection bandwidth

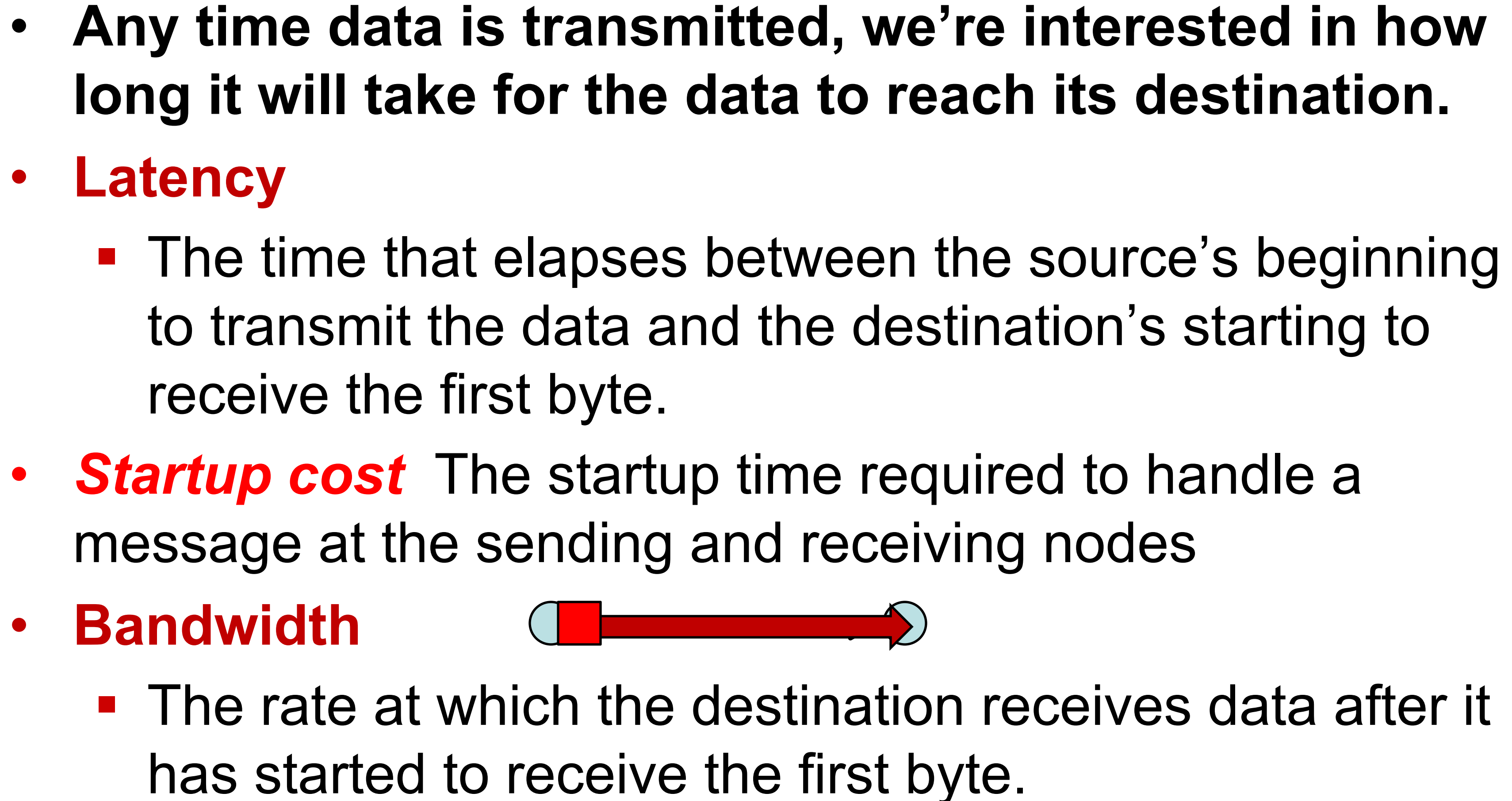
- **Example of bisection width**



- **Bisection bandwidth**
  - Sum bandwidth of links that cut the network into two equal halves.
  - Choose the minimum one.

More definitions on network performance

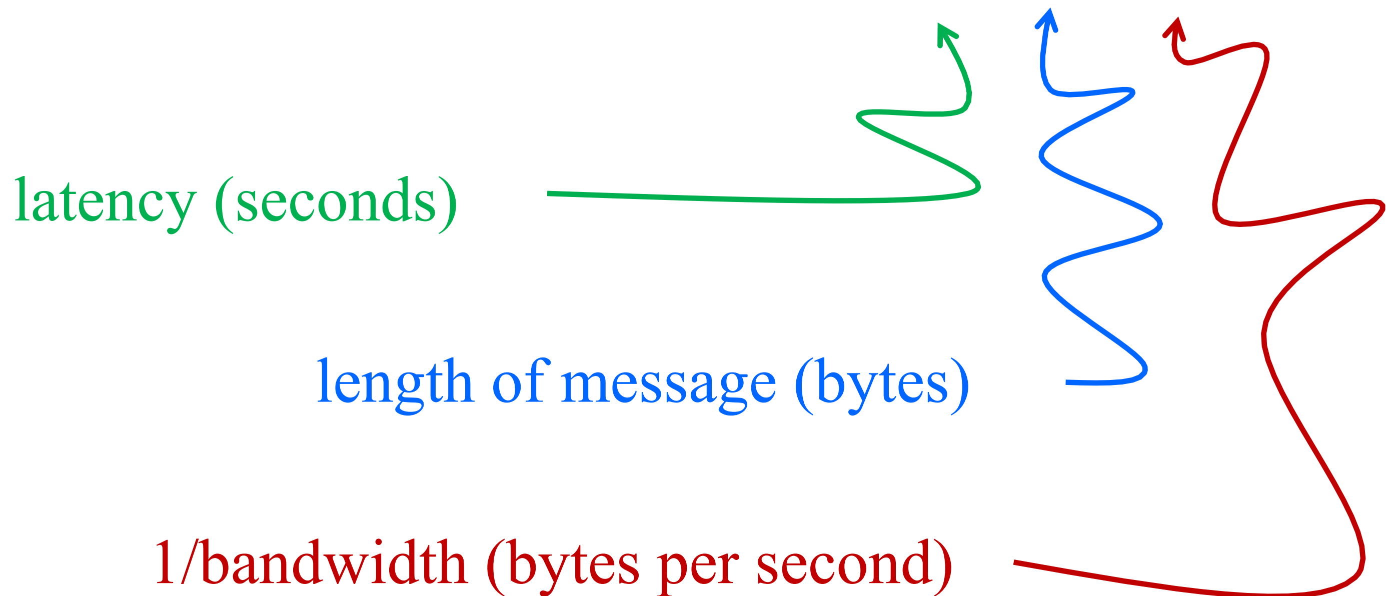
- **Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.**
- **Latency**
  - The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.
- ***Startup cost*** The startup time required to handle a message at the sending and receiving nodes
- **Bandwidth**



  - The rate at which the destination receives data after it has started to receive the first byte.

Network transmission cost

Message transmission time = $\alpha + m\,\beta$



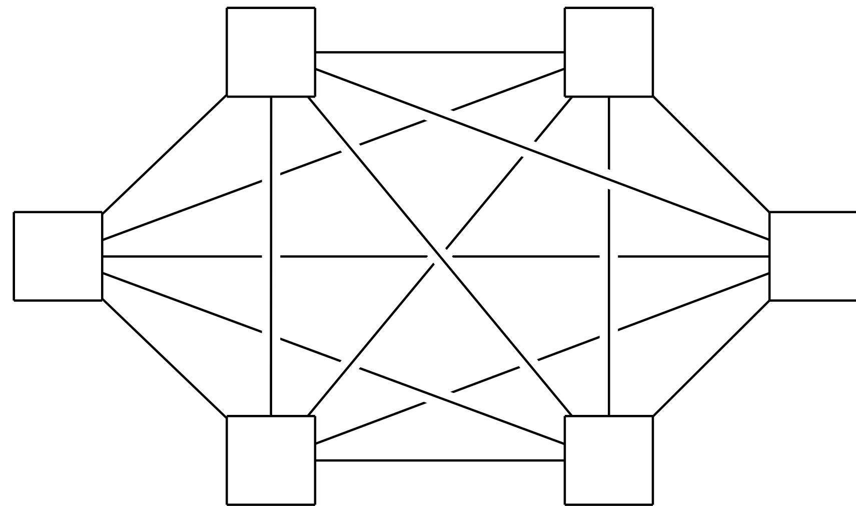
Typical latency/startup cost: Tens of microseconds ~ 1 millisecond

Typical bandwidth: 100 MB ~ 1GB per second

Fully connected network

- **Each switch is directly connected to every other switch.**

impractical



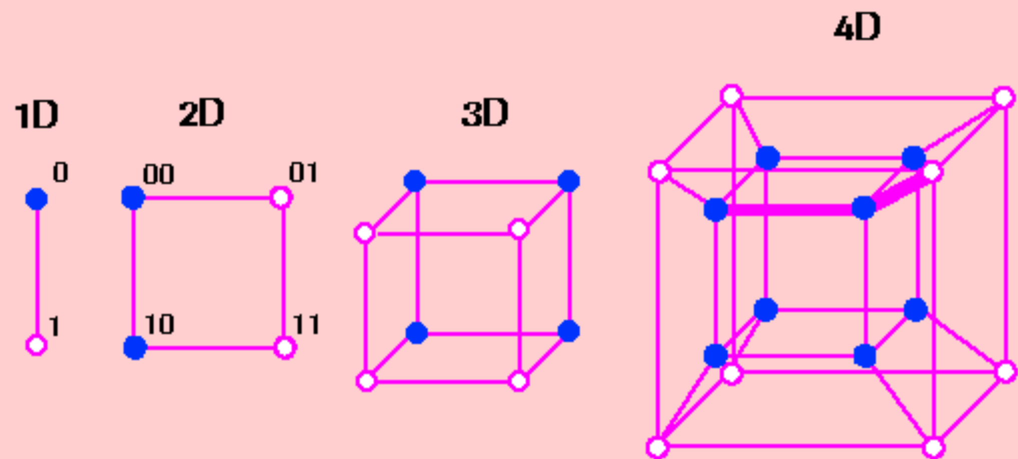
bisection width = $p^2/4$

Figure 2.11

Copyright © 2010, Elsevier Inc. All rights Reserved

Hypercube

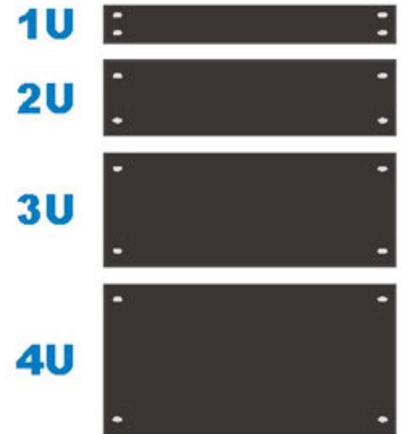
- **Built inductively:**
  - A one-dimensional hypercube is a fully-connected system with two processors.
  - A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches.
  - Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.

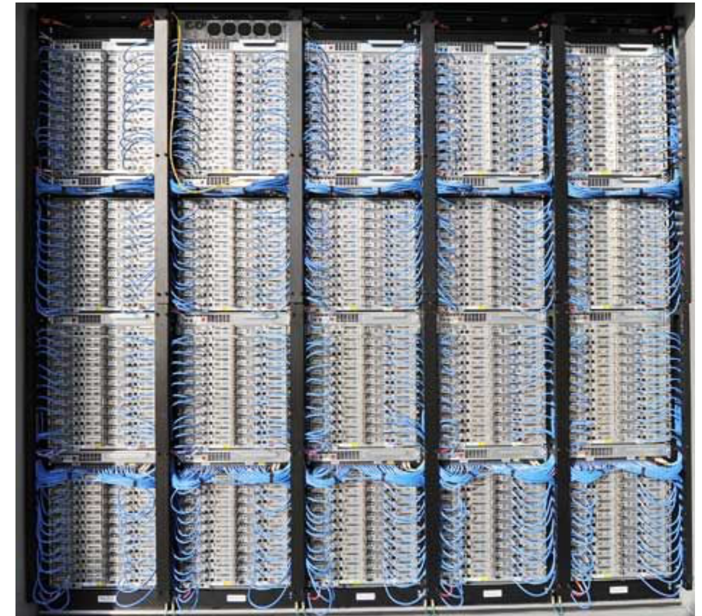


To construct an n-dimensional cube, copy an (n-1)-dimensional cube, then connect corresponding nodes in the original and the

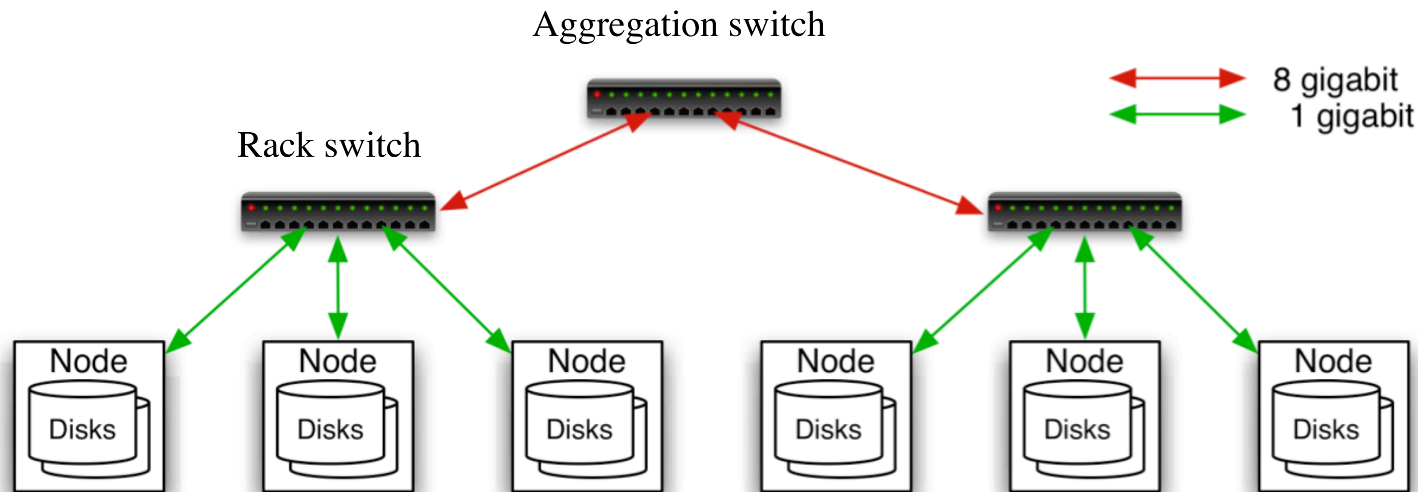
Commodity Computing Clusters

- **Use already available computing components**
  - Commodity servers, interconnection network, & storage
  - Less expensive while Upgradable with standardization
- **Great computing power at low cost**



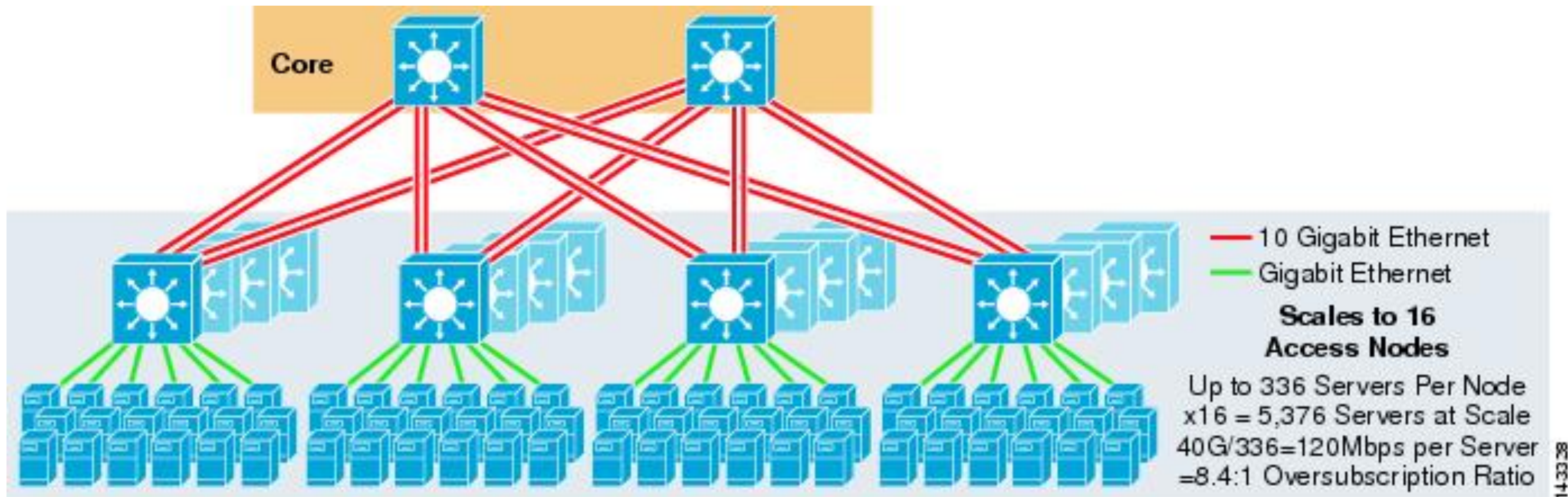


Typical network for a cluster



- **40 nodes/rack, 1000-4000 nodes in cluster**
- **1 Gbps bandwidth in rack, 8 Gbps out of rack**
- **Node specs :**
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks

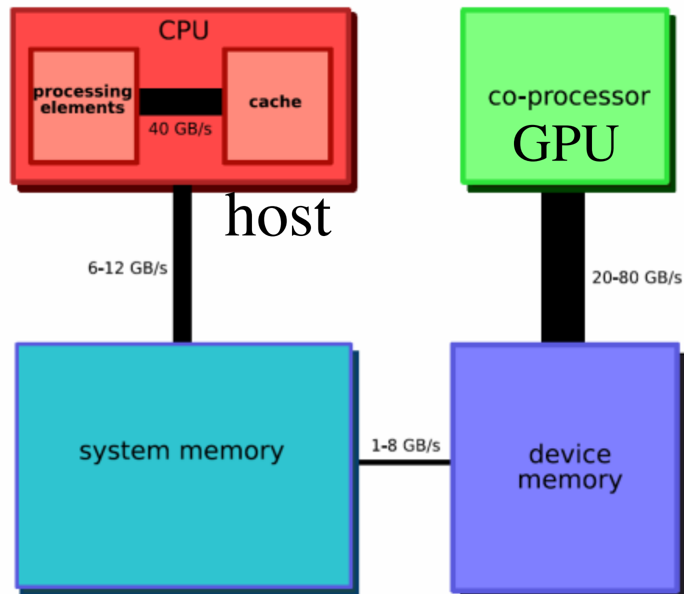
Layered Network in Clustered Machines

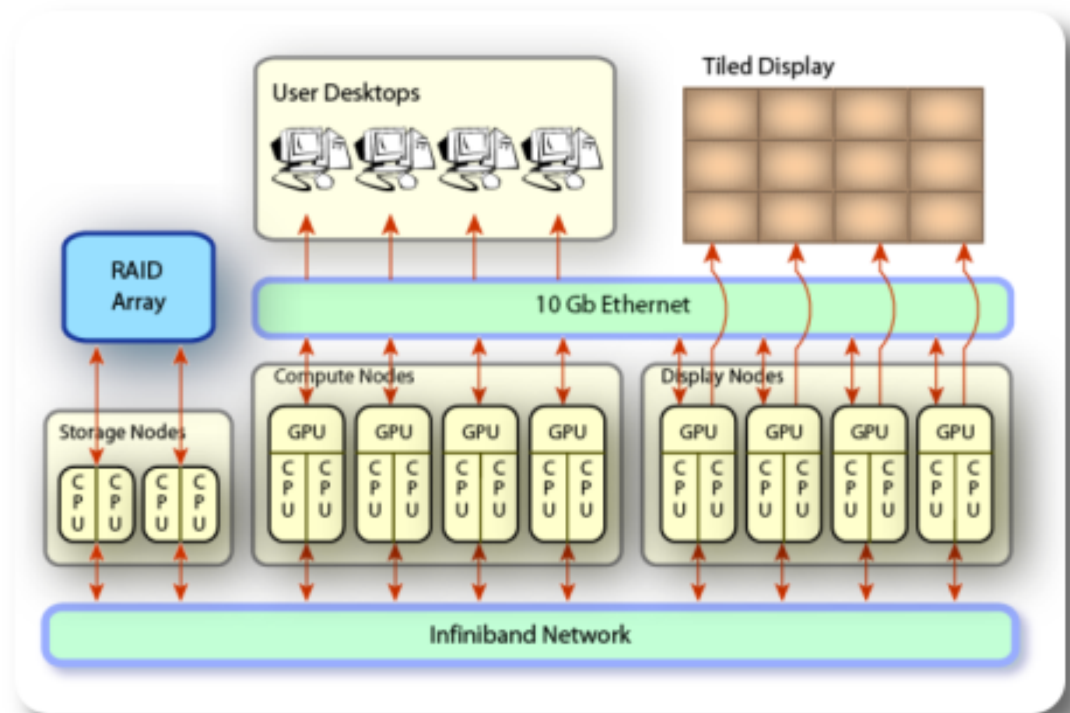


- **A layered example from Cisco: core, aggregation, the edge or top-of-rack switch.**
- **http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_3a.html**

Hybrid Clusters with GPU

Node in a CPU/GPU cluster

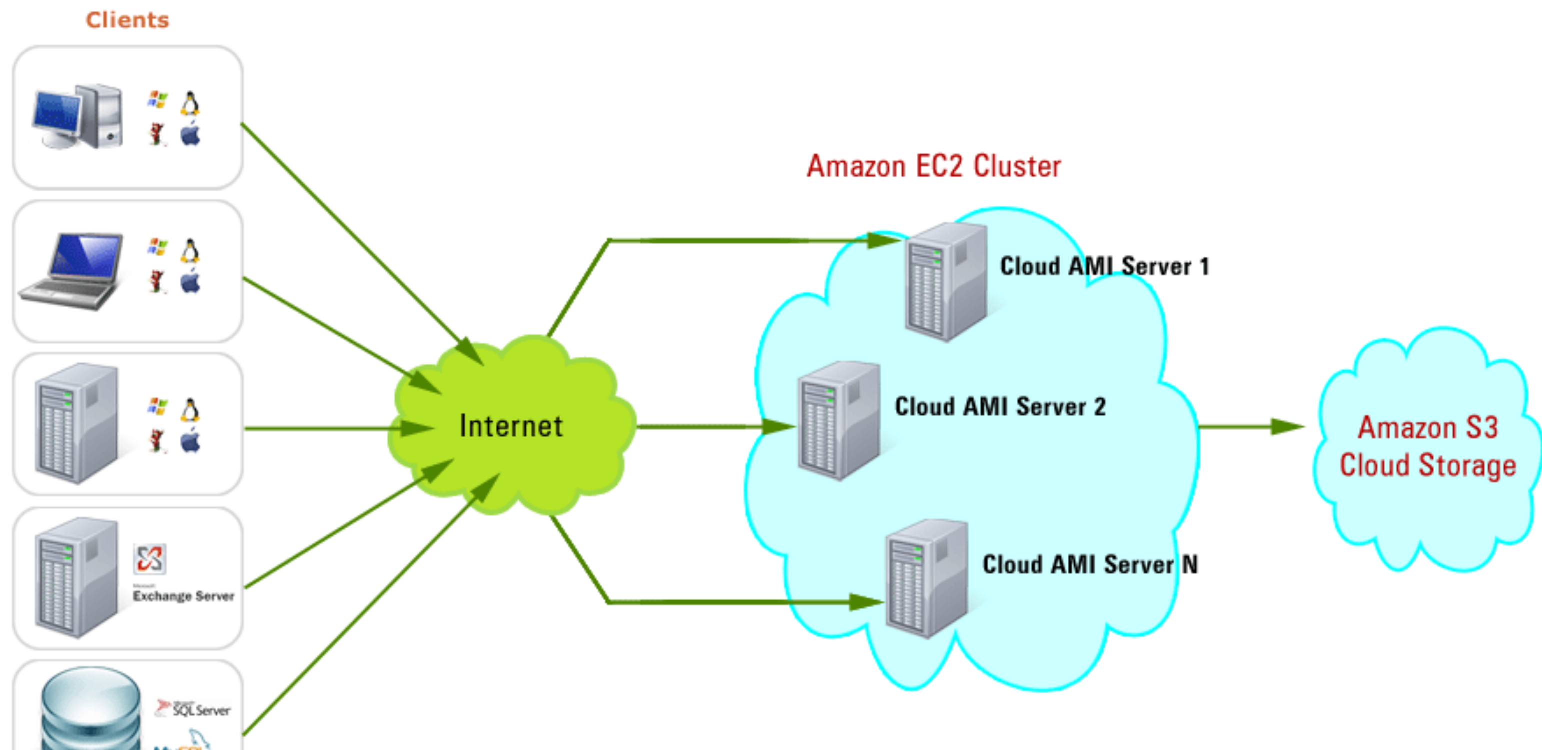




- A Maryland cluster couples CPUs, GPUs, displays, and storage.
- Applications in visual and scientific computing

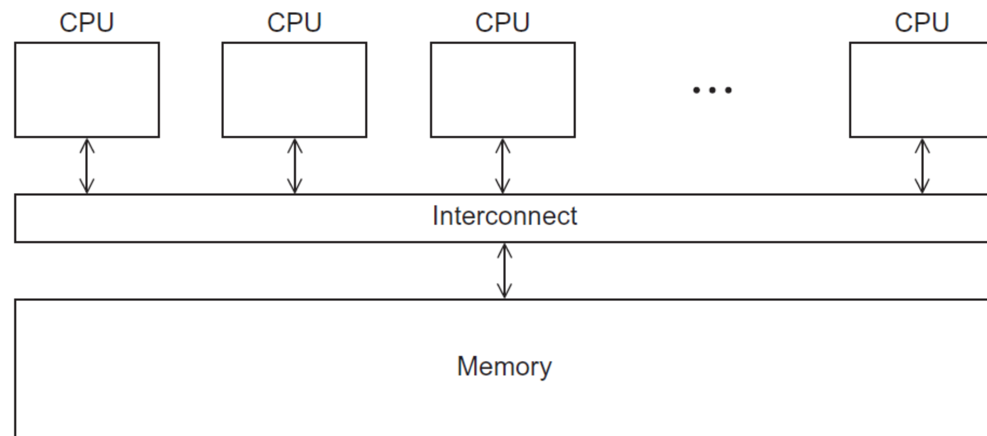
Cloud Computing with Amazon EC2

- **On-demand elastic computing**
  - Allocate a Linux or windows cluster only when you need.
    - Pay based on time usage of computing instance/storage
  - Expandable or shrinkable



Usage Examples with Amazon EC2

- **A 32-node, 64-GPU cluster with 8TB storage**
  - Each node is a AWS computing instance extended with 2 Nvidia M2050 GPUs, 22 GB of memory, and a 10Gbps Ethernet interconnect.
  - **$82/hour to operate (**based on Cycle Computing blog)
    - Annual cost in 2011: 82*8*52=$34,112. Cheaper today.
    - **Otherwise:** ~$150+K to purchase + datacenter cost.

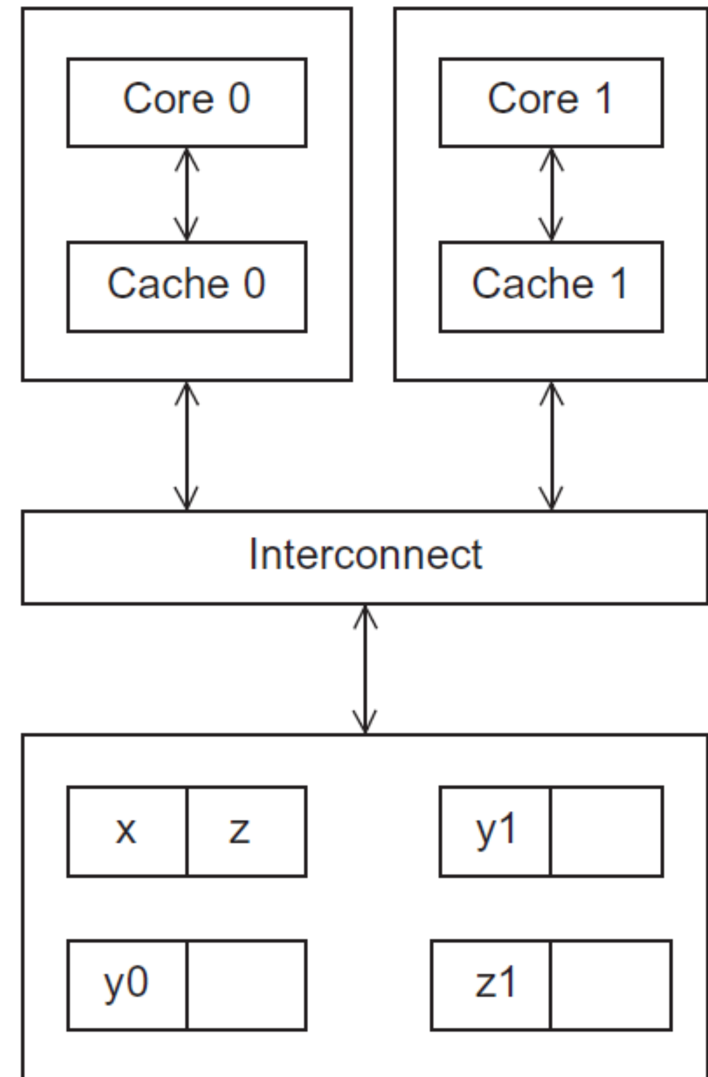


SHARED MEMORY ARCHITECTURE WITH CACHE COHERENCE

Copyright © 2010, Elsevier Inc. All rights Reserved

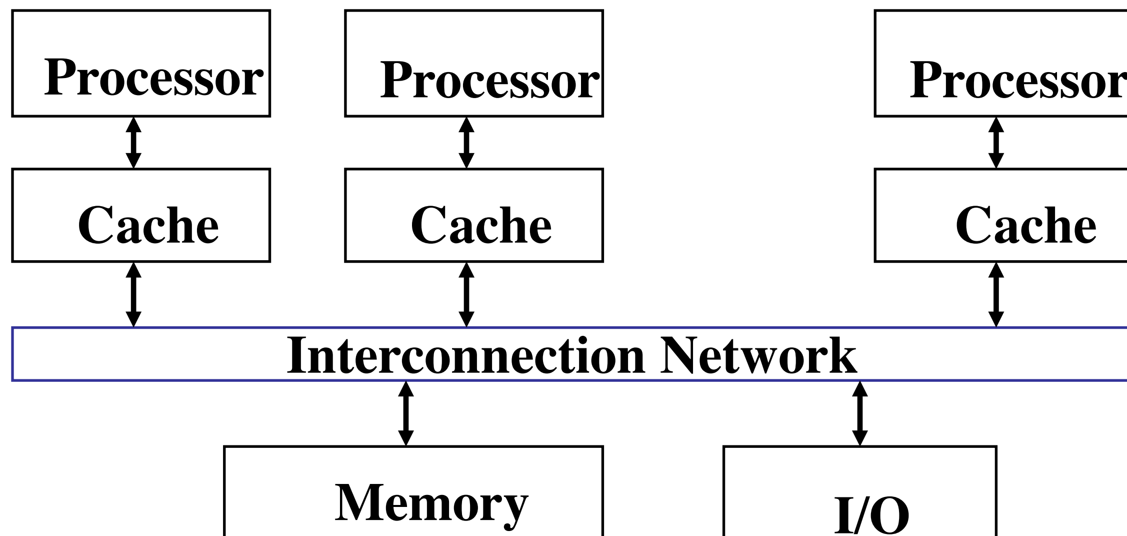
Cache coherence

- **Programmers have no control over caches and when they get updated.**
- **Hardware makes cache updated cache coherently**
  - Snooping bus
  - Directory-based



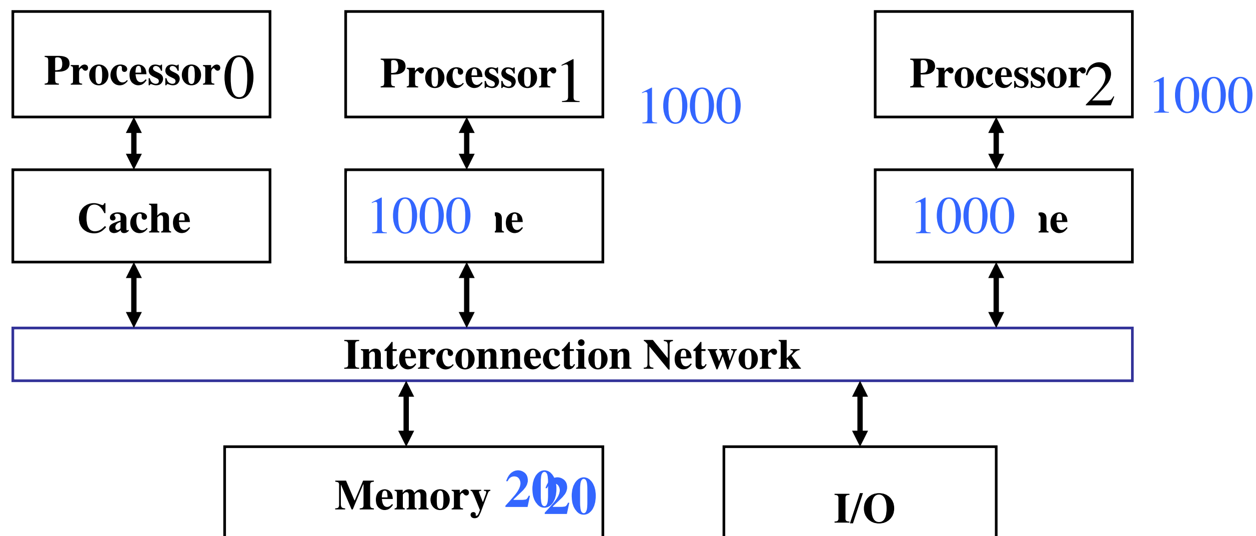
Shared Memory Architecture with Cache Coherence

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory



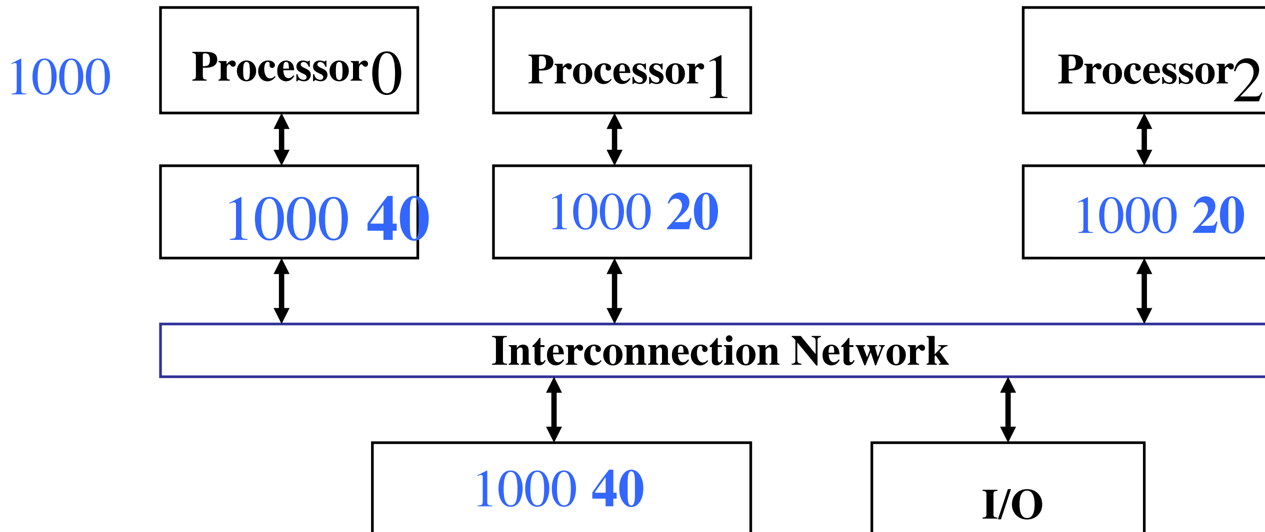
Shared Memory and Caches

- **What if?**
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- **Now:**
  - Processor 0 writes Memory[1000] with 40



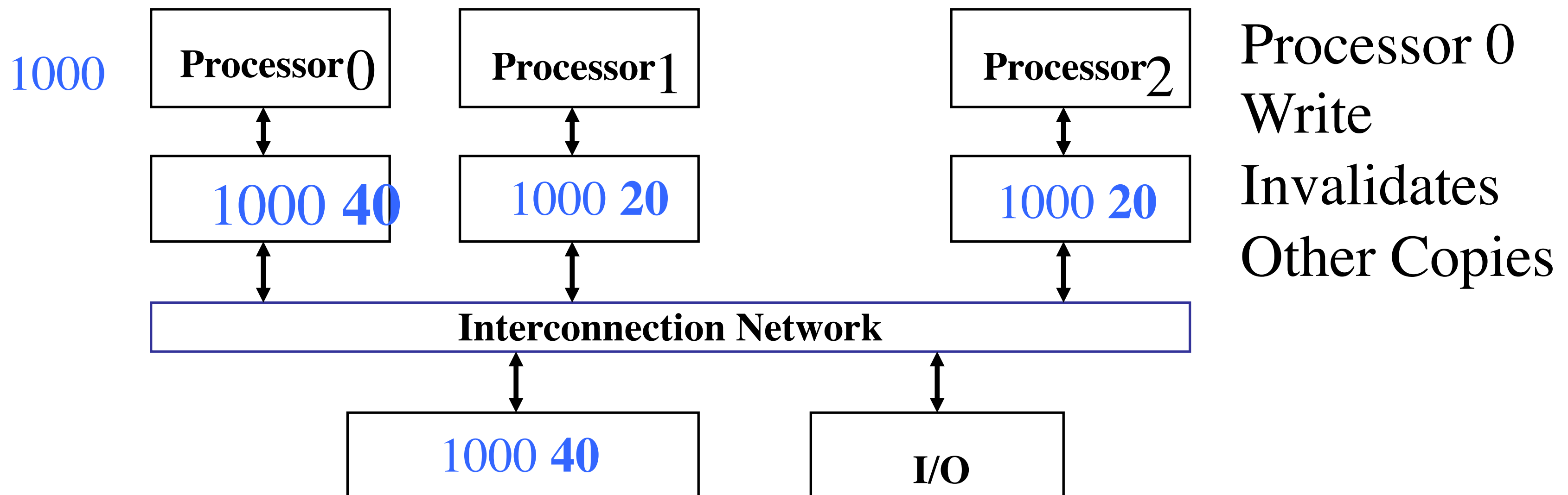
Problem?

Keeping Multiple Caches Coherent

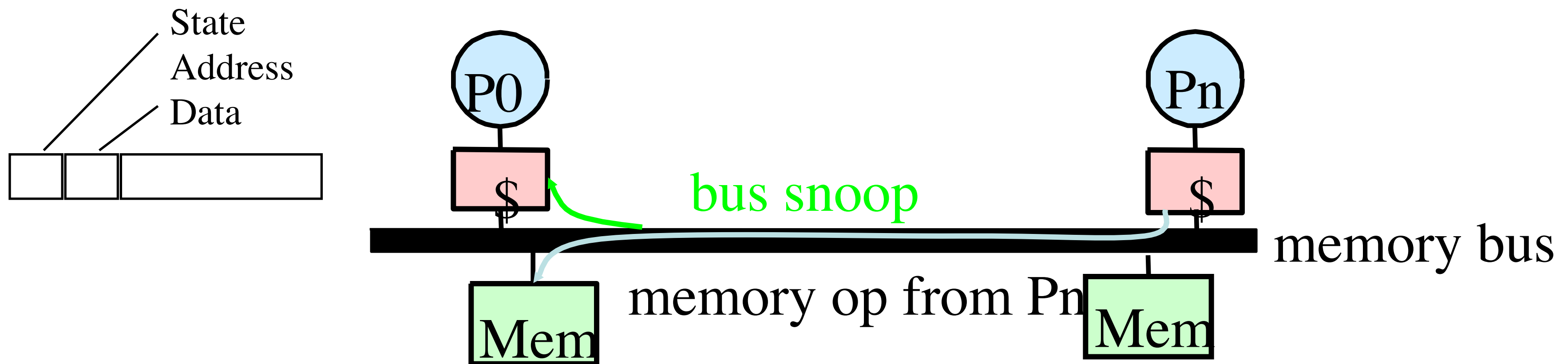
- **Architect's job: shared memory => keep cache values coherent**
- **Idea: When any processor has cache miss or writes, notify other processors via interconnection network**
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- **Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold**
  - Invalidate any copies of same address modified in other cache

Shared Memory and Caches

- **Example, now with cache coherence**
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40

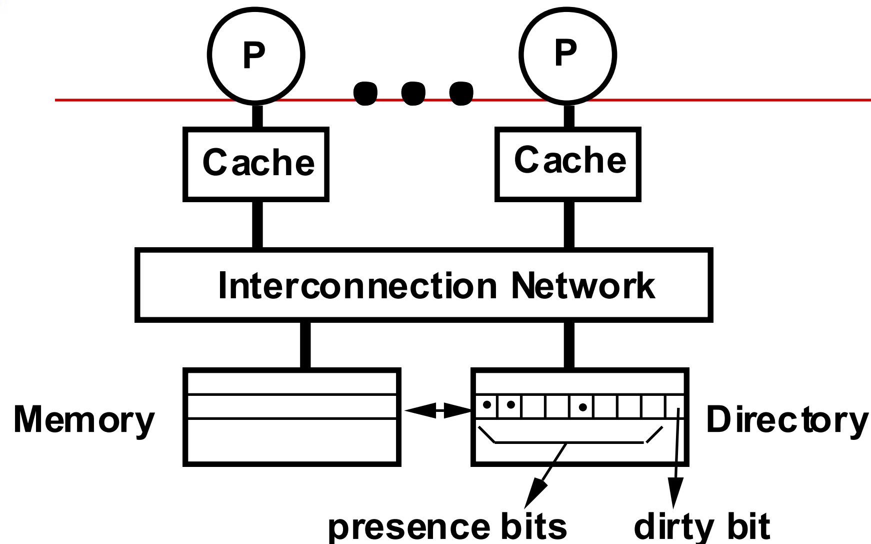


Snoopy Cache-Coherence Protocols



- **Memory bus is a broadcast medium**
- **Caches contain information on which addresses they store**
- **Cache Controller "snoops" all transactions on the bus**
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - Take action to ensure coherence
    - invalidate, update, or supply value
  - Many possible designs
- **Not scalable for a large number of processors**

Scalable Shared Memory: Directories



- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- **Every memory block has associated directory information**
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- **Each Reader recorded in directory**
- **Processor asks permission of memory before writing:**
  - Send invalidation to each cache with read-only copy
  - Wait for acknowledgements before returning permission for writes

Slide source: John Kubiatowicz

Directory Based Memory/Cache Coherence

- **Keep Directory to keep track of which memory stores latest copy of data. Meta information:**
  - Valid/invalid. Dirty (inconsistent with memory). Shared
- **When a processor executes a write operation to shared data, basic design choices are:**
  - With respect to memory:
    - Write through cache: do the write in memory as well as cache
    - Write back cache: wait and do the write later, when the item is flushed
  - With respect to other cached copies
    - Update: give all other processors the new value
    - Invalidate: all other processors remove from cache

Intuitive Memory Model

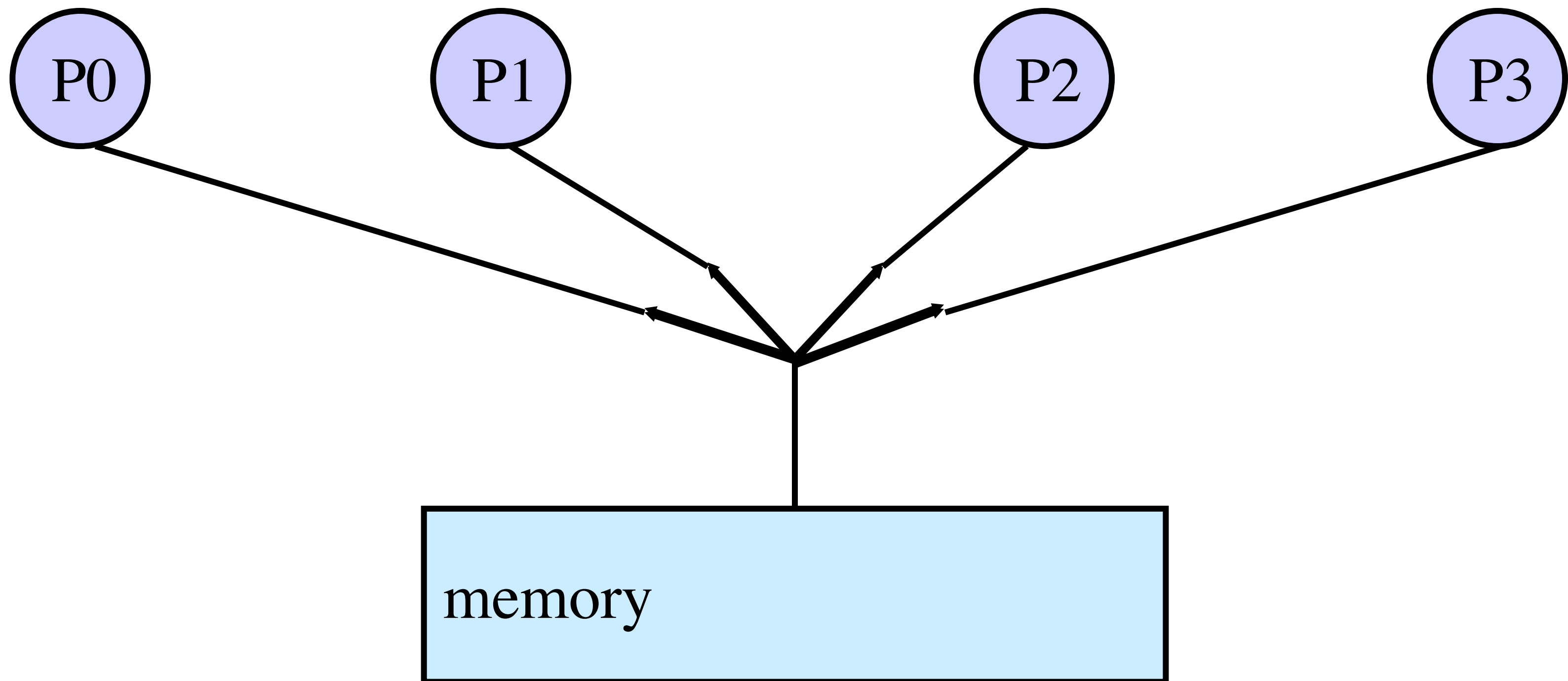
- **Reading an address should return the last value written to that address**
- **Easy in uniprocessors**
  - except for I/O
- **Multiprocessor cache coherence problem is more pervasive and more performance critical**
- **More formally, this is called sequential consistency:**

"A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Source:Demmel/Yellick

Sequential Consistency Intuition

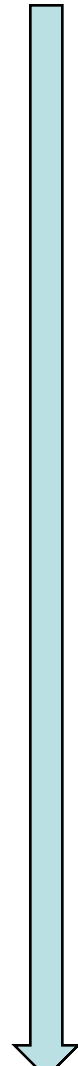
- Sequential consistency says the machine ***behaves as if*** it does the following



Sequential Consistency Example

Processor 1	Processor 2
LD_1 A ⇒ 5	LD_5 B ⇒ 2
LD_2 B ⇒ 7	…
ST_1 A,6	LD_6 A ⇒ 6
…	ST_4 B,21
LD_3 A ⇒ 6	…
LD_4 B ⇒ 21	LD_7 A ⇒ 6
ST_2 B,13	…
ST_3 B,4	LD_8 B ⇒ 4

One Consistent Serial Order



LD_1 A ⇒ 5
LD_2 B ⇒ 7
LD_5 B ⇒ 2
ST_1 A,6
LD_6 A ⇒ 6
ST_4 B,21
LD_3 A ⇒ 6
LD_4 B ⇒ 21
LD_7 A ⇒ 6
ST_2 B,13
ST_3 B,4
LD_8 B ⇒ 4

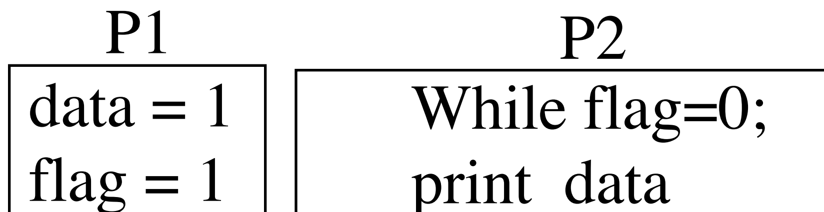
Slide source: John Kubiatowicz

Memory Consistency Semantics

What does this imply about program behavior?

- **No process ever sees "garbage" values. Processors always see values written by some processor**
- **The value seen is constrained by program order on all processors**
  - **Time always moves forward**
- **Example: *spin lock***
  - **P1 writes data=1, then writes flag=1**
  - **P2 waits until flag=1, then reads data**

initially: flag=0
data=0



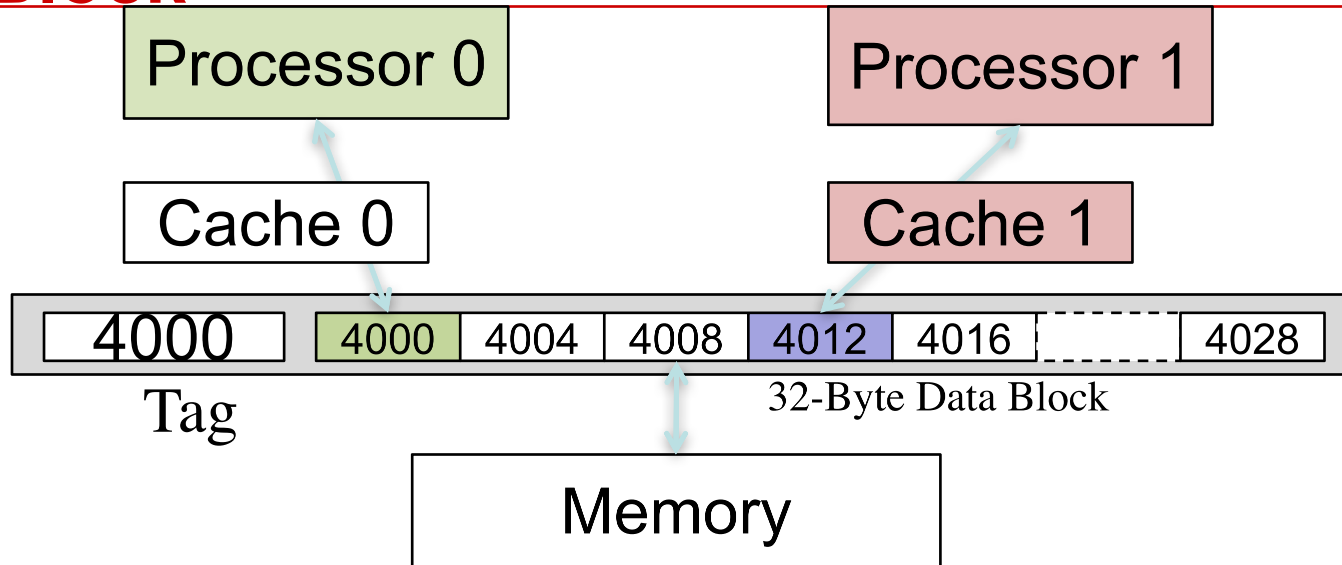
If P2 sees the new value of flag (=1), it must see the new value of data (=1)

If P2 reads flag	Then P2 may read data
0	1
0	0
1	1

Cache Coherence and Sequential Consistency

- **HW/SW features may break sequential consistency (SC):**
  - The compiler reorders/removes code.
  - Write buffers (place to store writes while waiting to complete)
    - Processors may reorder writes to merge addresses (not FIFO)
    - Write X=1, Y=1, X=2 (second write to X may happen before Y's)
  - Prefetch instructions cause read reordering (read data before flag)
  - The network reorders the two write messages. .
- **Some commercial systems give up SC**
  - A correct program on a SC processor may be incorrect on one that is not

False Sharing: Cache Coherency Tracked by Block



- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

False Sharing

- **Block ping-pongs between two caches even though processors are accessing disjoint variables**
  - Shared data is modified by multiple processors.
  - Multiple processors update data within the same cache line.
  - This updating occurs very frequently (for example, in a tight loop).
- **Effect called *false sharing***
  - *Cause cache miss for every write, even they write to different locations.*
- **How can you prevent it?**
  - Let parallel iterations write to different cache blocks (as much as possible)
    - allocate data used by each processor contiguously, or at least avoid interleaving in memory
  - Make use of private data as much as possible

PERFORMANCE

Copyright © 2010, Elsevier
Inc. All rights Reserved

Speedup

- **Number of cores = p**
- **Serial run-time = T_{serial}**
- **Parallel run-time = $T_{parallel}$**

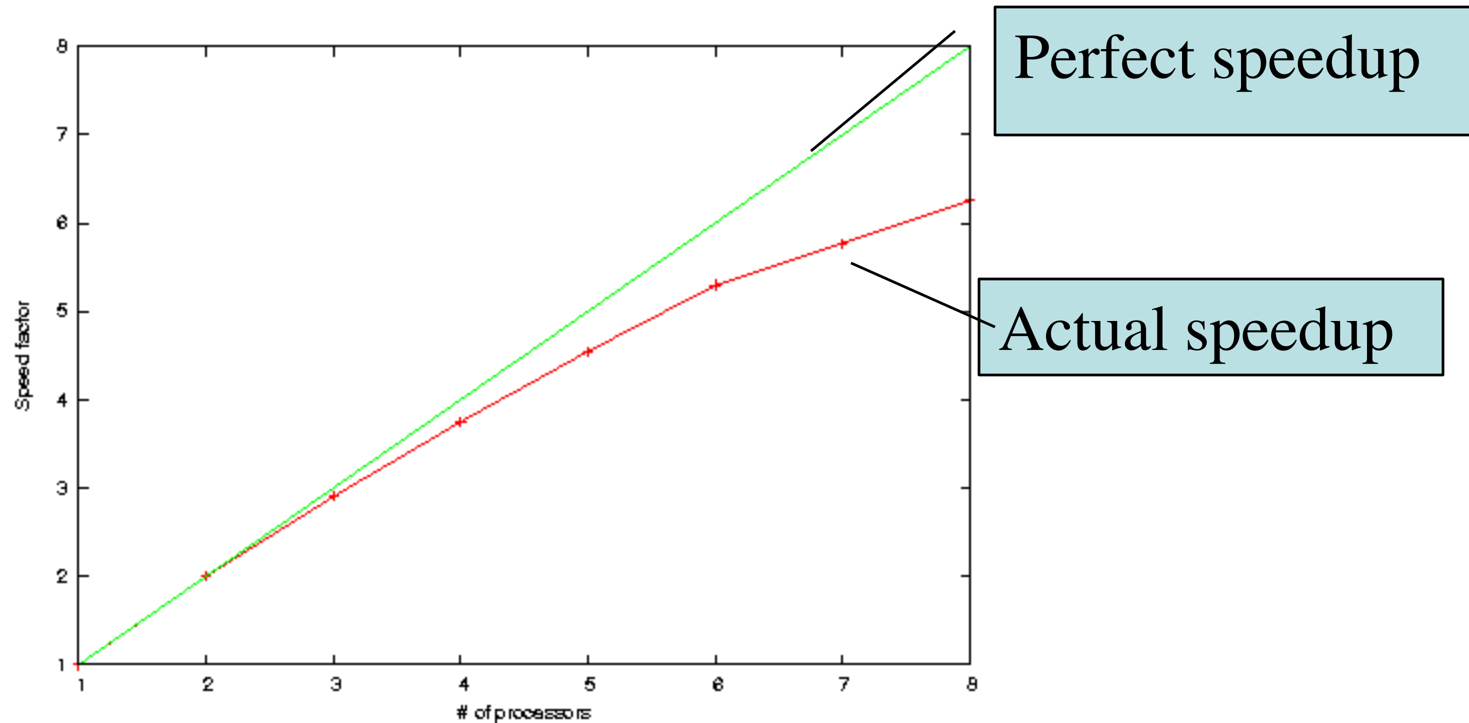
If linear speedup

$$T_{parallel} = T_{serial} / p$$

Copyright © 2010, Elsevier Inc. All rights Reserved

Speedup of a parallel program

$$S = \frac{T_{serial}}{T_{parallel}}$$



Speedup Graph Interpretation

- *Linear speedup*
  - Speedup proportionally increases as p increases
- *Perfect linear speedup*
  - Speedup =p
- *Superlinear speedup*
  - Speedup >p
  - It is not possible in theory.
  - It is possible in practice
    - Data in sequential code does not fit into memory.
    - Parallel code divides data into many machines and they fit into memory.

Copyright © 2010, Elsevier Inc. All rights Reserved

Efficiency of a parallel program

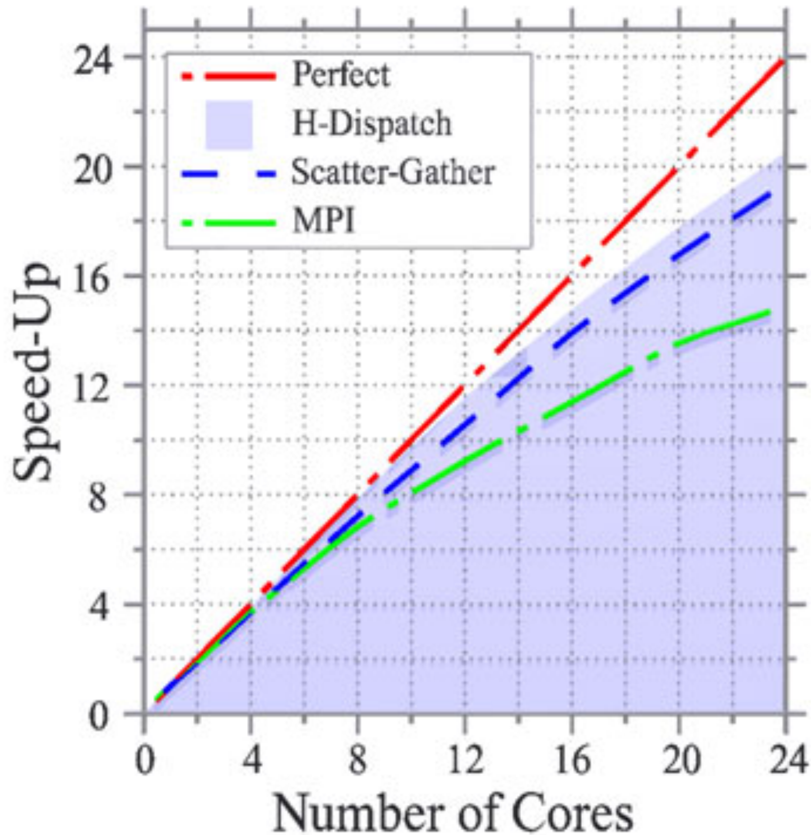
$$E = \frac{\text{Speedup}}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

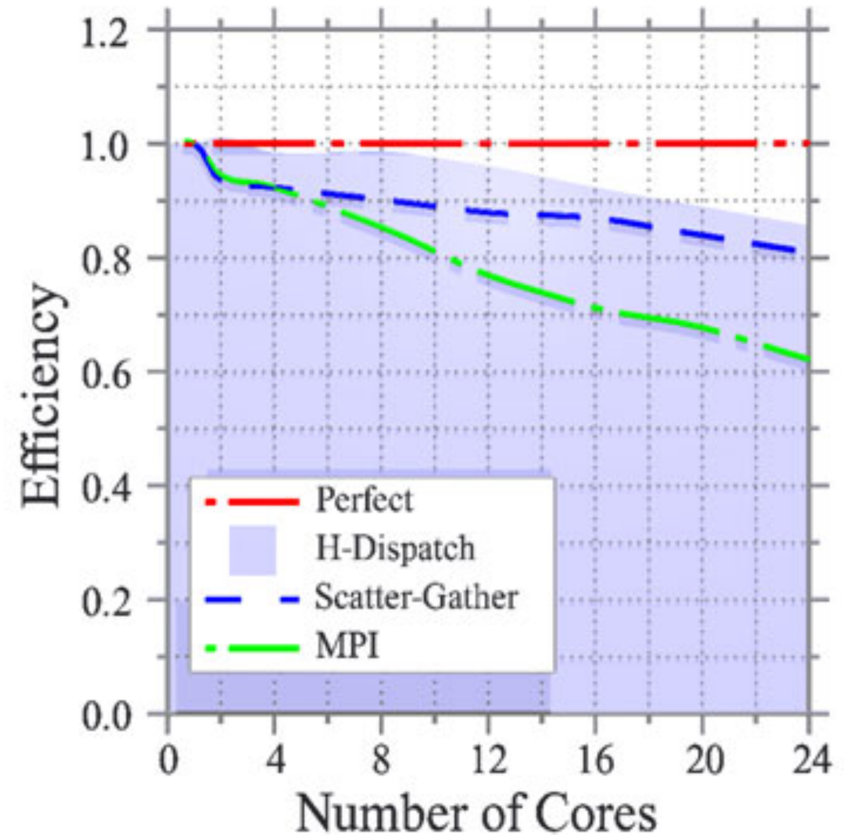
Measure how well-utilized the processors are, compared to effort wasted in communication and synchronization.

Example:

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

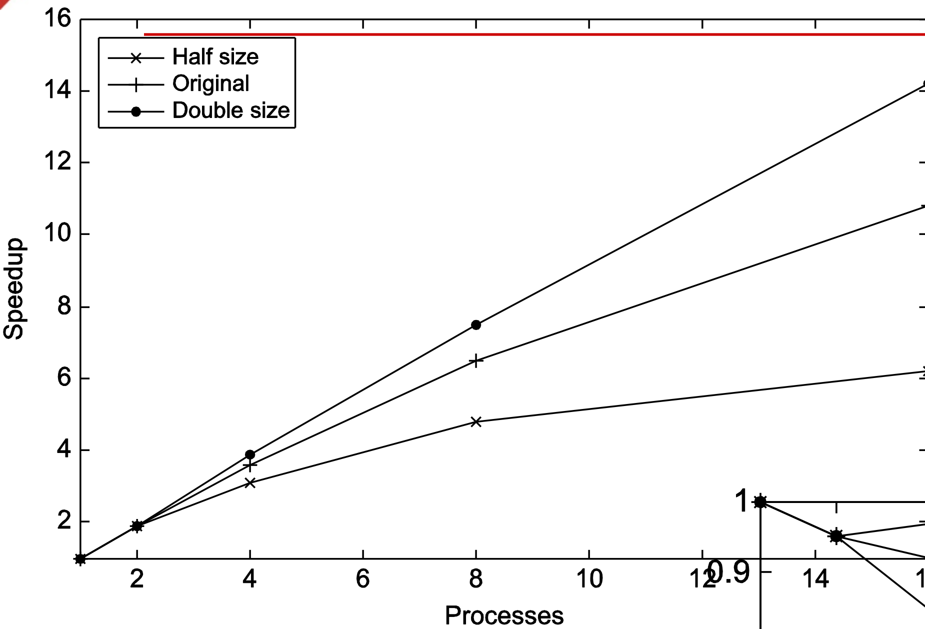
Typical speedup and efficiency of parallel code



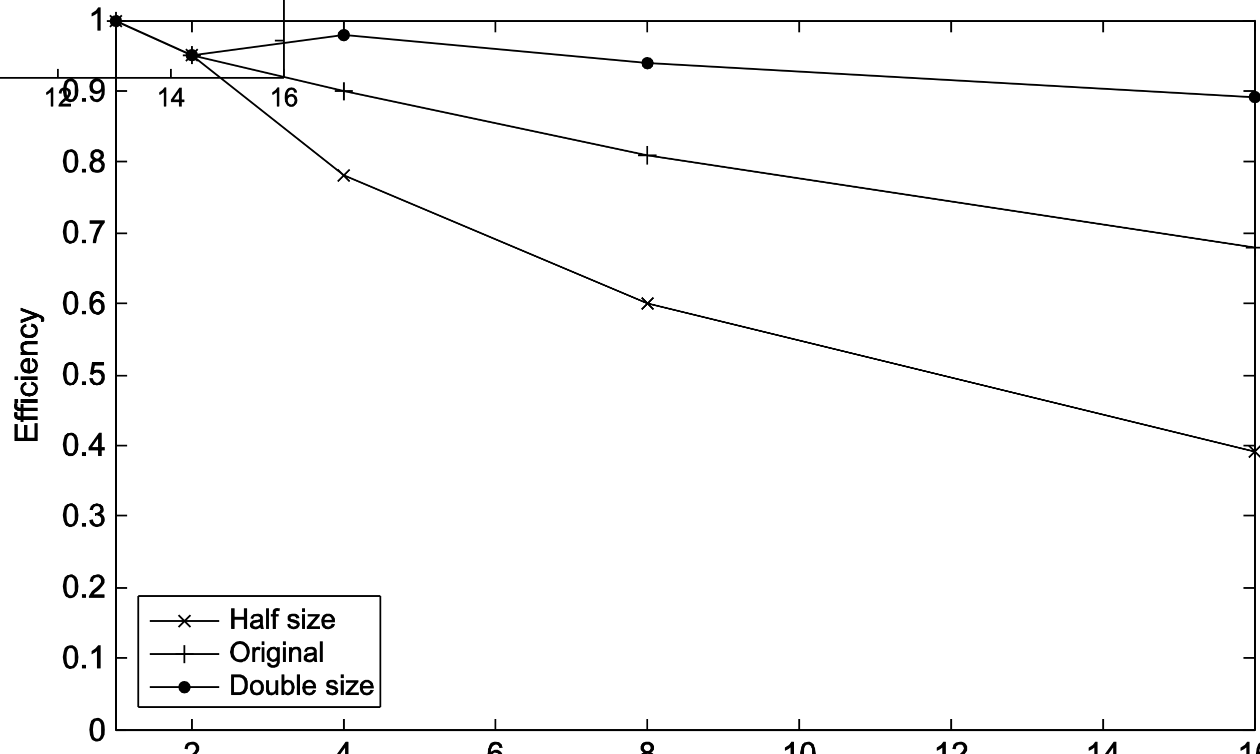


Efficiency often goes down as more cores are used

Problem Size Impact on Speedup and Efficiency

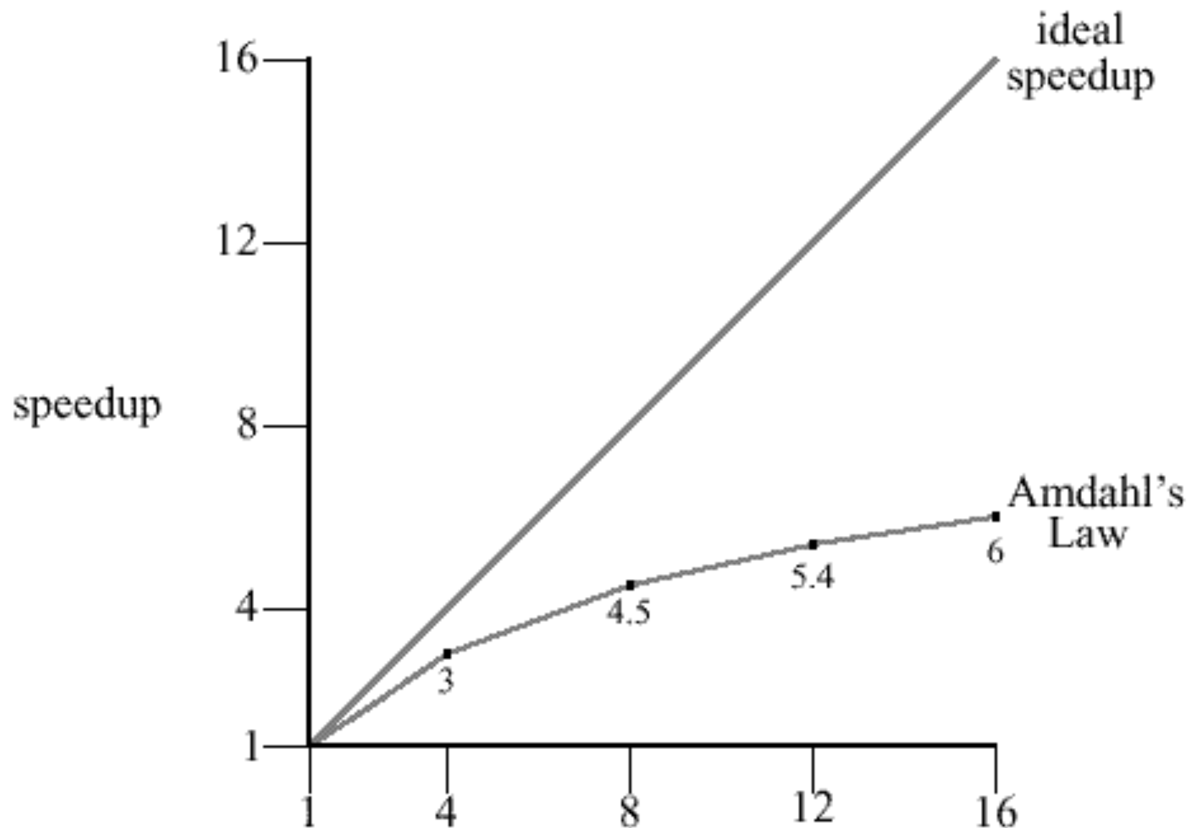


Efficiency often increases when increasing problem size



Amdahl Law: Limitation of Parallel Performance

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be limited — regardless of the number of cores available.



Copyright © 2010, Elsevier Inc. All rights Reserved

Finding Enough Parallelism

- **Suppose only part of an application seems parallel**
- **Amdahl's law**
  - let x be the fraction of work done sequentially, so (1-x) is fraction parallelizable
  - P = number of processors

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$\le 1/(x + (1-x)/P)$$

$$\le 1/x$$

- Even if the parallel part speeds up perfectly performance is limited by the sequential part

Example of Amdahl's Law

- Example:
  - We can parallelize 90% of a serial program.
  - Parallelization is "perfect" regardless of the number of cores *p* we use.
  - T_{serial} = 20 seconds
  - Runtime of parallelizable part is $0.9 \times T_{serial} / p = 18 / p$
  - Runtime of "unparallelizable" part is $0.1 \times T_{serial} = 2$

Overall parallel run-time is

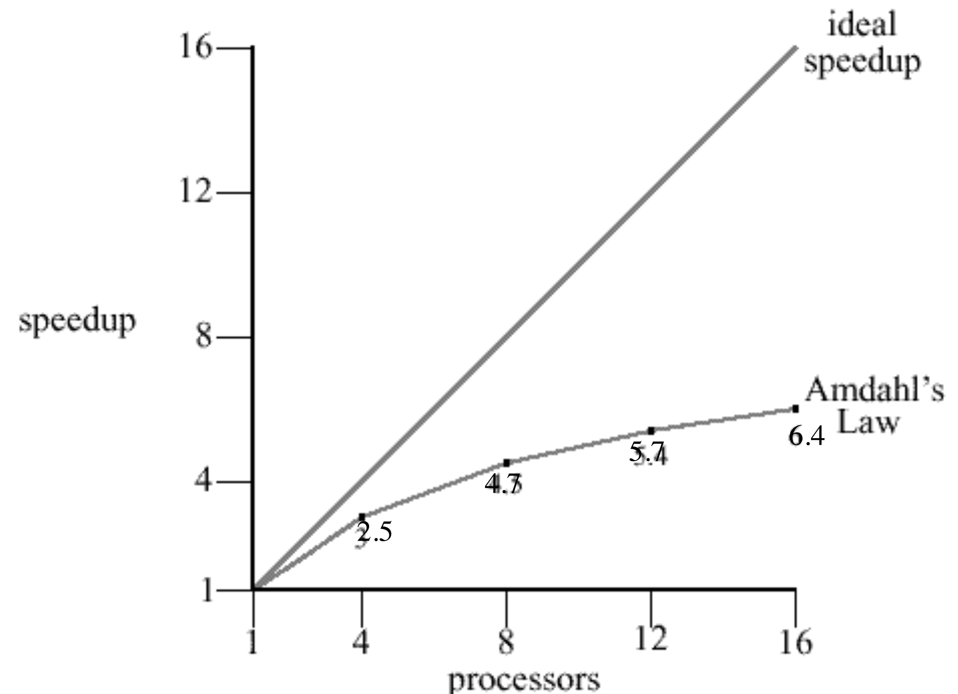
- $T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$

Example (cont.)

- **Speed up**

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

- **S < 20/2 =10**



How to measure sequential and parallel time?

- **What time?**
  - CPU time vs wall clock time
- **A program segment of interest?**
  - Setup startup time
  - Measure finish time

Copyright © 2010, Elsevier Inc. All rights Reserved

Taking Timings for a Code Segment

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Taking Timings

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Copyright © 2010, Elsevier
Inc. All rights Reserved

Measure parallel time with a barrier

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

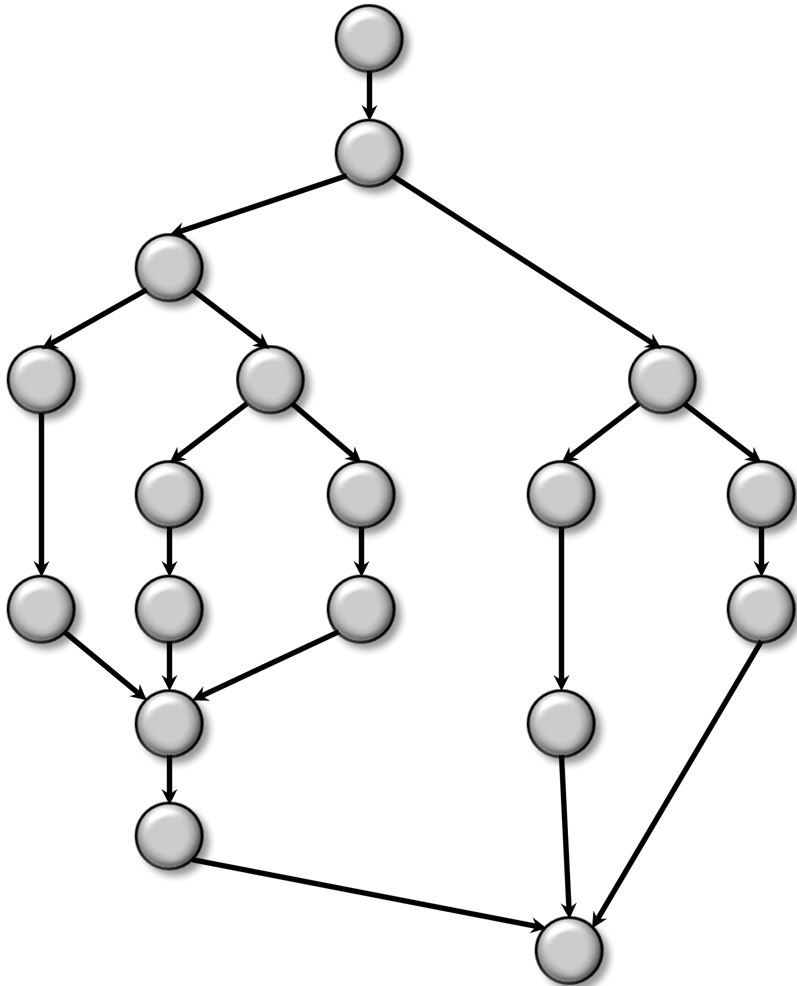
Several possible performance models

- **Execution time and parallelism:**
  - Work / Span Model with directed acyclic graph
- **Detailed models that try to capture time for moving data:**
  - Latency / Bandwidth Model for message-passing
  - Disk IO
- **Model computation with memory access (for hierarchical memory)**
- **Other detailed models we won't discuss: LogP, ….**

  - From John Gibert's 240A course

Model Parallelism using a Directed Acyclic Graph

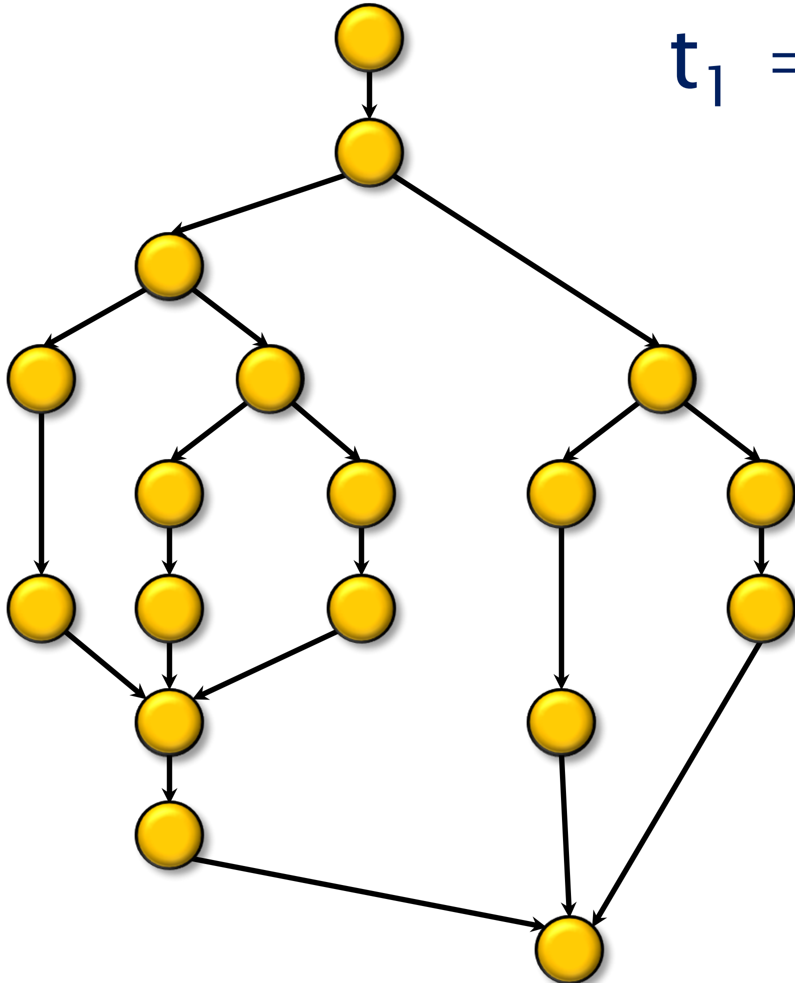
t_p = execution time on p processors



Work / Span Model

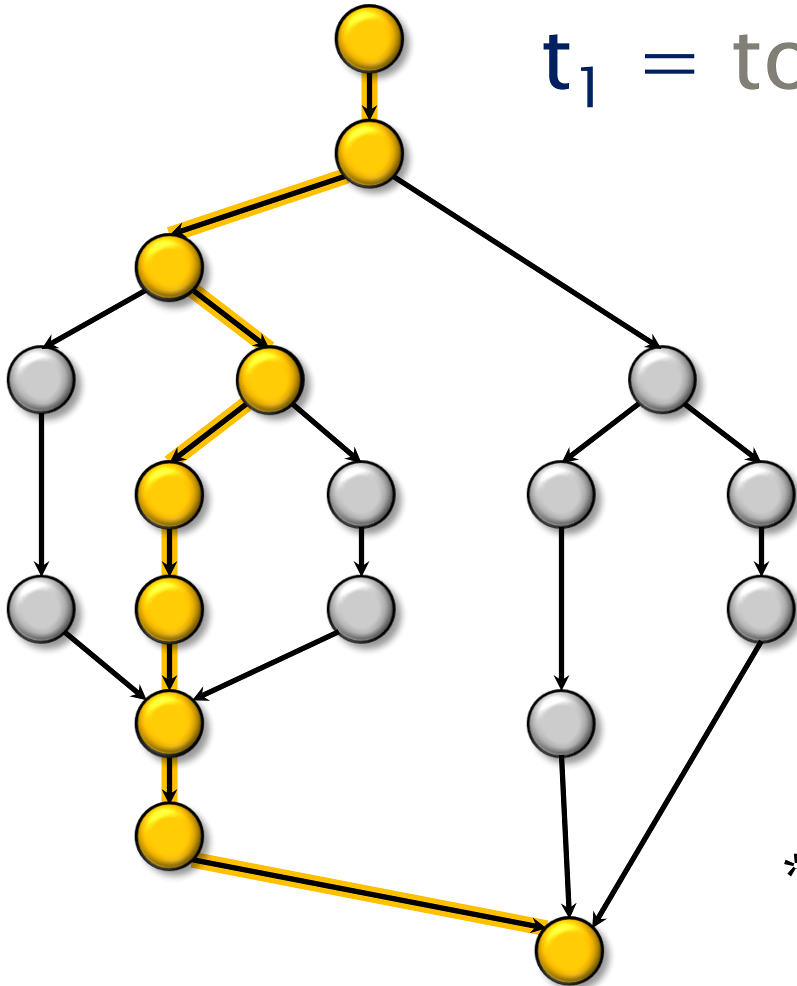
t_p = execution time on p processors

t_1 = ***work***



Work / Span Model

t_p = execution time on p processors

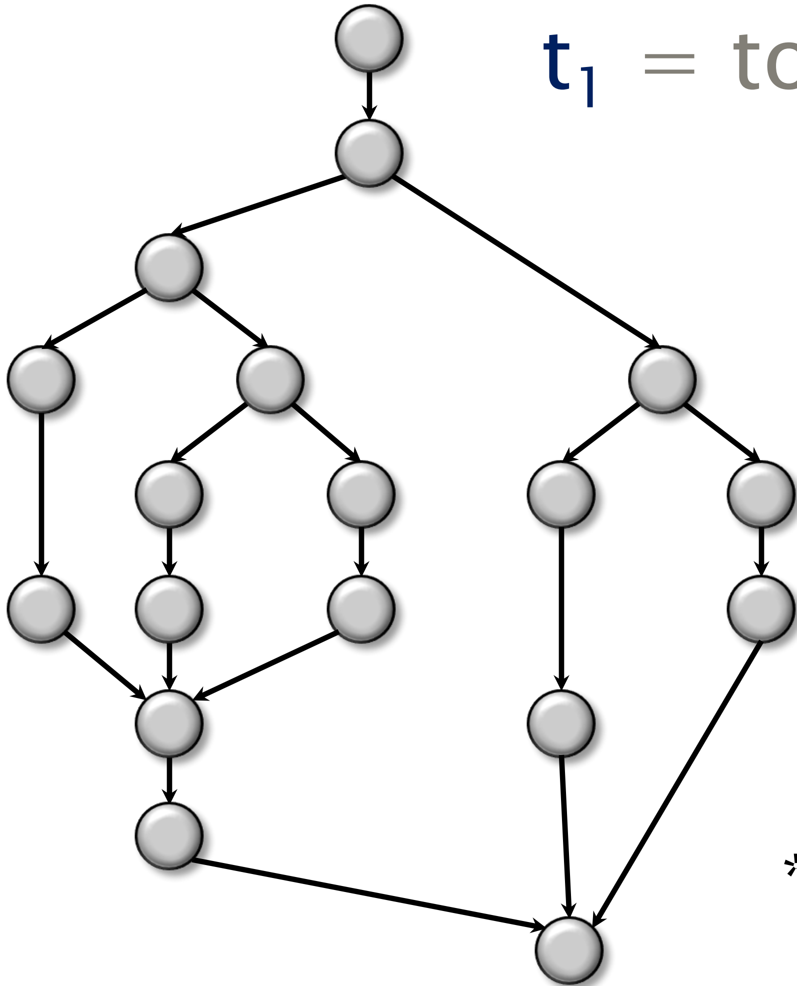


t_1 = total ***work*** t_∞ = ***span*** *

*Also called ***critical-path length*** or ***computational depth***.

Work / Span Model

t_p = execution time on p processors



t_1 = total ***work*** t_∞ = ***span*** *

WORK LAW

- $t_p \geq t_1/p$

SPAN LAW

- $t_p \geq t_\infty$

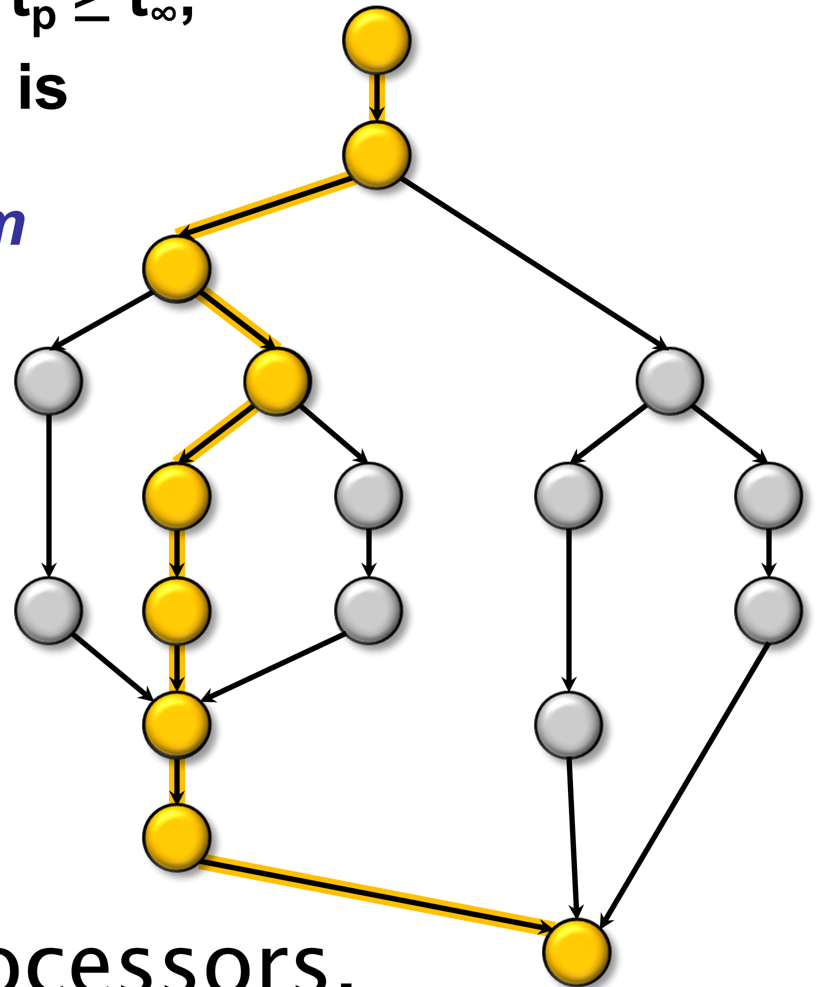
*Also called ***critical-path length*** or ***computational depth***.

Potential Parallelism

Because the Span Law requires $t_p \geq t_\infty$, the maximum possible speedup is

t_1/t_∞ = *(potential) parallelism*

= the average amount of work per step along the span.



Note

t_1 / t_P = ***speedup*** on p processors.

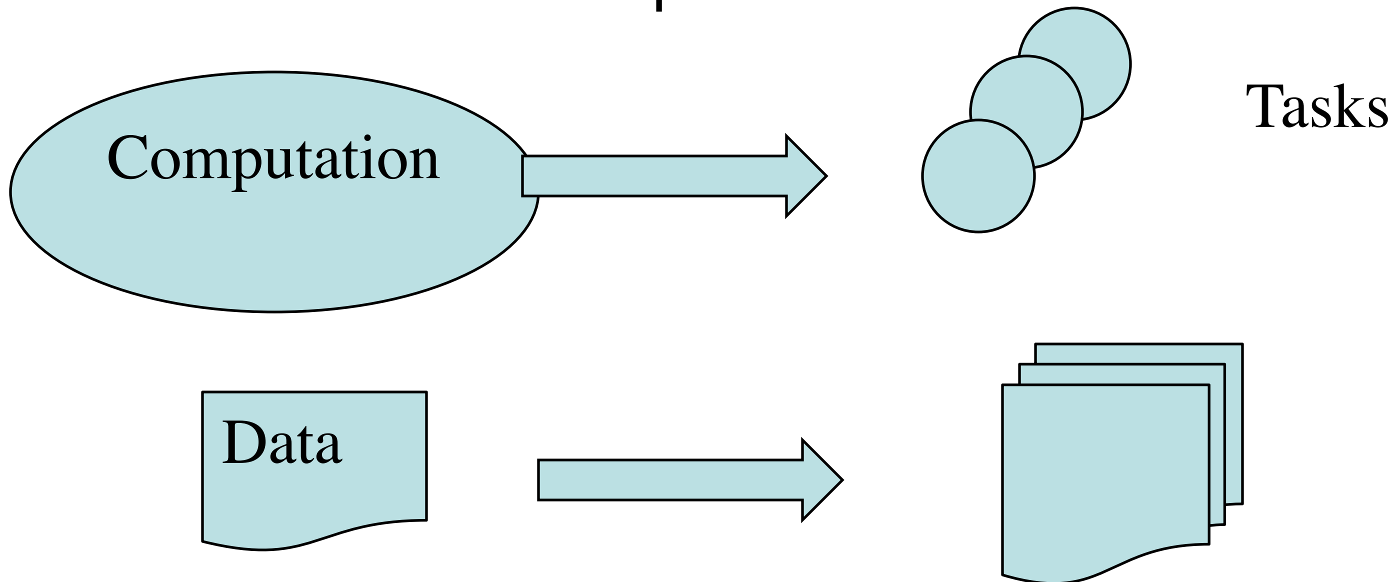
PARALLEL PROGRAM DESIGN

Copyright © 2010, Elsevier Inc. All rights Reserved

Foster's methodology: 4-stage design

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

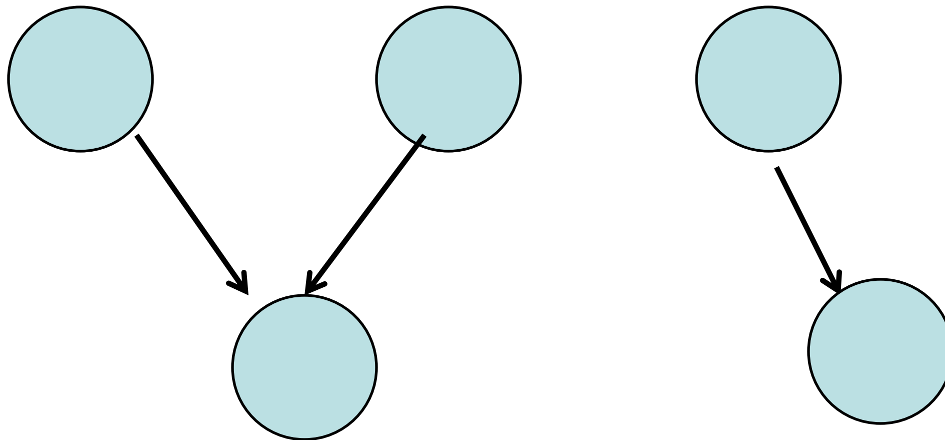
   The focus here should be on identifying tasks that can be executed in parallel.



Copyright © 2010, Elsevier Inc. All rights Reserved

Foster's methodology

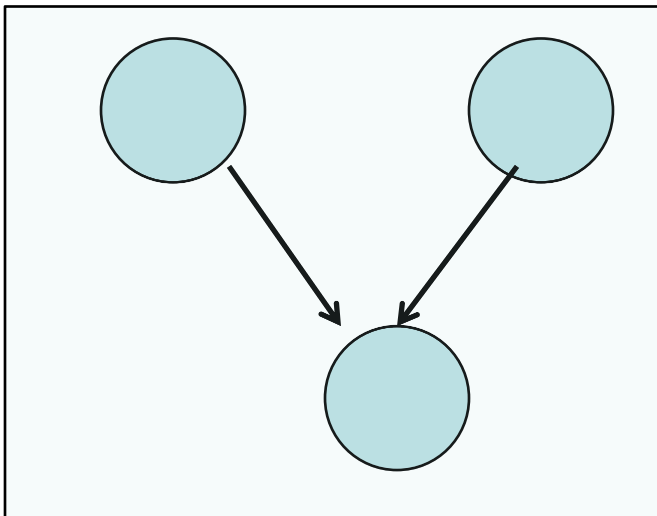
2. Communication:
   - Identify dependence among tasks
   - Determine inter-task communication

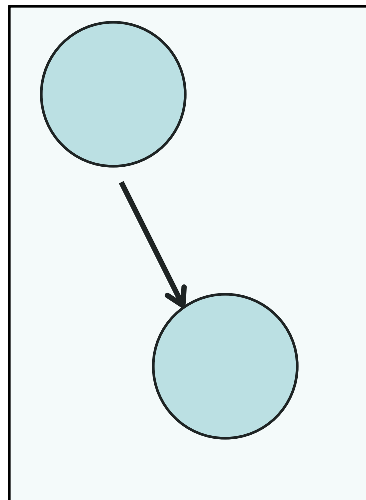


Copyright © 2010, Elsevier
Inc. All rights Reserved

Foster's methodology

3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.
   - Reduce communication overhead →Coarse grain tasks
   - May reduce parallelism sometime



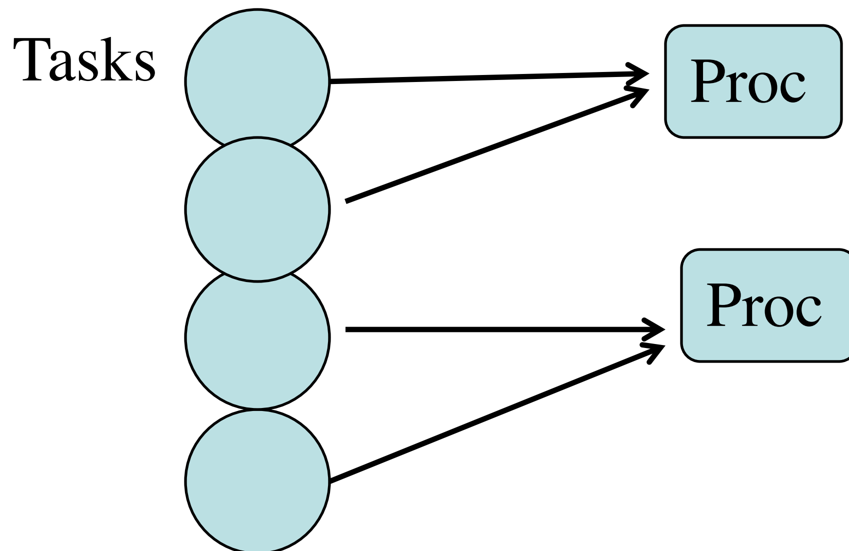


Copyright © 2010, Elsevier Inc. All rights Reserved

Foster's methodology

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.



Copyright © 2010, Elsevier Inc. All rights Reserved

Input and Output in Parallel Programs

- **Two options for I/O**
  - Option 1:
    - In distributed memory programs, only process 0 will access *stdin*.
    - In shared memory programs, only the master thread or thread 0 will access *stdin*.
  - Option 2:
    - all the processes/threads can access *stdout* and *stderr*.
- Because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

Copyright © 2010, Elsevier Inc. All rights Reserved

Input and Output: Practical Strategies

- Debug output should always include the rank or id of the process/thread that's generating the output.

- Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

- fflush(stdout) may be necessary to ensure output is not delayed when order is important.
  - printf("hello \n"); fflush(stdout);

Concluding Remarks (1)

- **Parallel hardware**
  - Shared memory and distributed memory architectures
  - Network topology for interconnect
- **Parallel software**
  - We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.
  - SPMD programs.

Copyright © 2010, Elsevier Inc. All rights Reserved

Concluding Remarks (2)

- **Input and Output**
  - One process or thread can access stdin, and all processes can access stdout and stderr.
    - However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing stdout.
- **Performance**
  - Speedup/Efficiency
  - Amdahl's law
  - Scalability
- **Parallel Program Design**
  - Foster's methodology

Copyright © 2010, Elsevier Inc. All rights Reserved