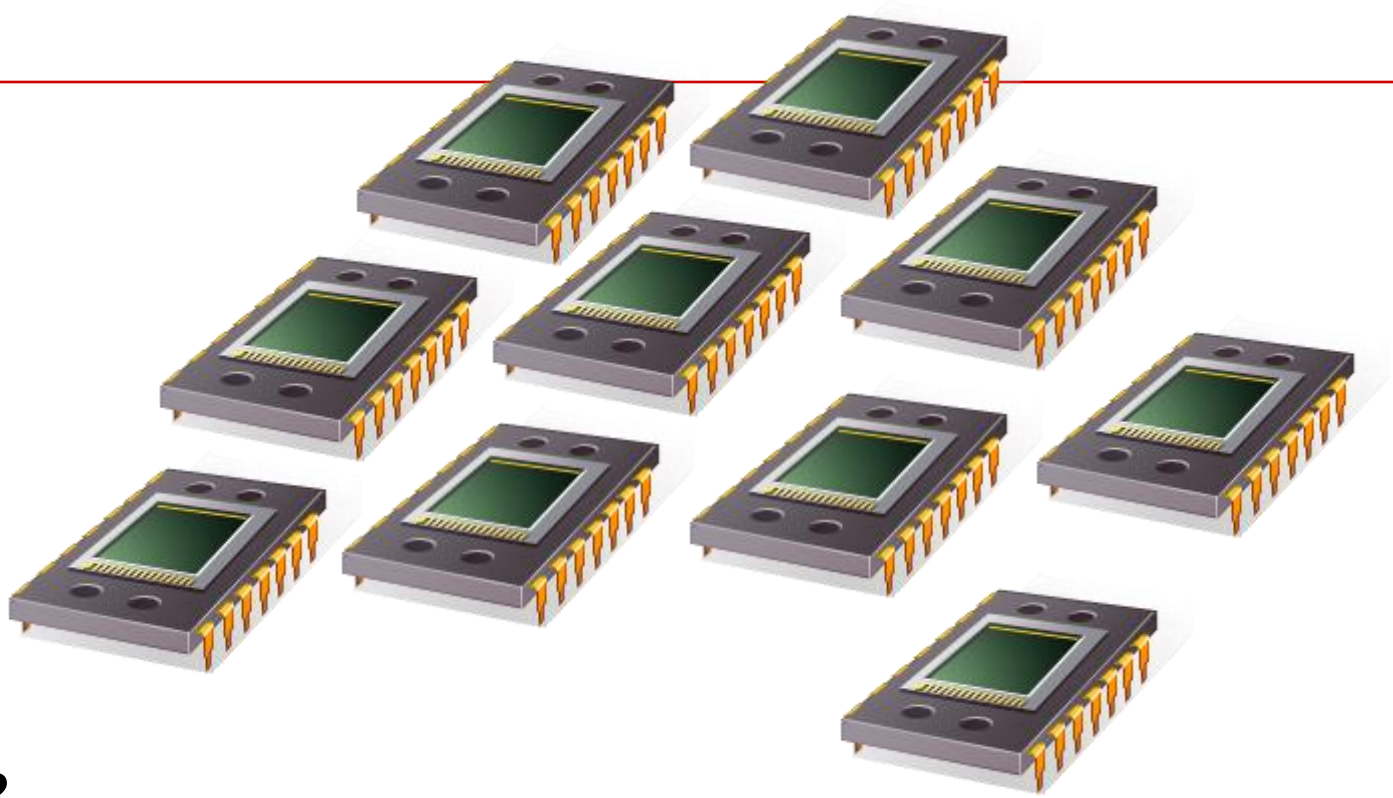


Chapter 2

Parallel Architecture, Software And Performance

UCSB CS140, T. Yang, 2014
Modified from texbook slides

Roadmap

- Parallel hardware
- Parallel software
- Input and output
- Performance
- 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

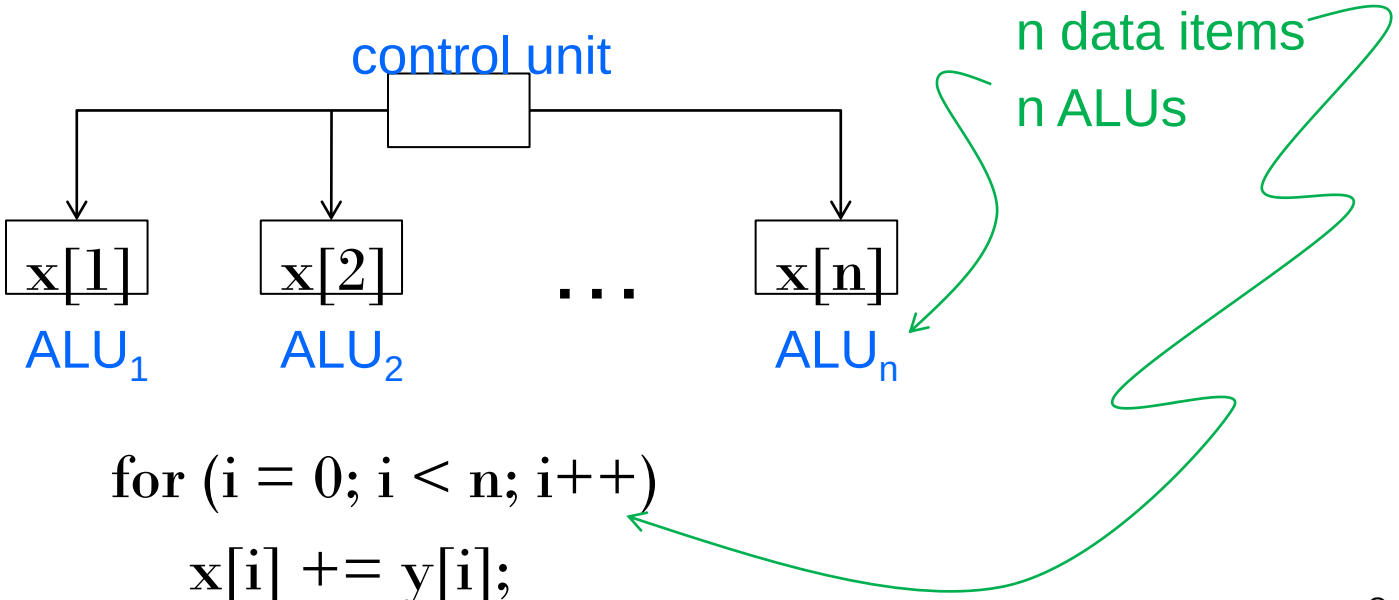
classic von Neumann

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

- What if we don't have as many ALUs (Arithmetic Logic Units) as data items?
- Divide the work and process iteratively.
- Ex. m = 4 ALUs (arithmetic logic unit) and n = 15 data items.

Round	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

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

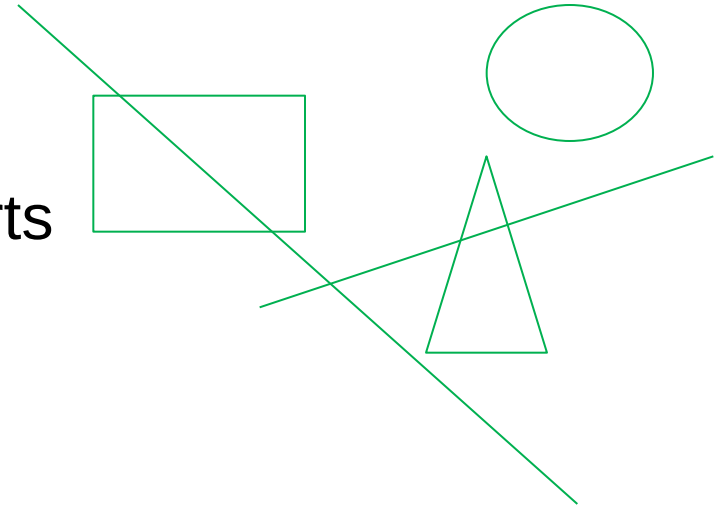
- **Pros**
  - Fast. Easy to use.
  - Vectorizing compilers are good at identifying code to exploit.
    - Compilers also can provide information about code that cannot be vectorized.
    - Helps the programmer re-evaluate code.
  - High memory bandwidth. Use every item in a cache line.

- **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)

- Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.



- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.
  - Several stages of this pipeline (called shader functions) are programmable.
  - Typically just a few lines of C code.

Copyright © 2010, Elsevier Inc. All rights Reserved

GPUs

- Shader functions are also implicitly 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.
- **Market shares:**

  Intel: 62% NVIDIA 17%, AMD. 21%

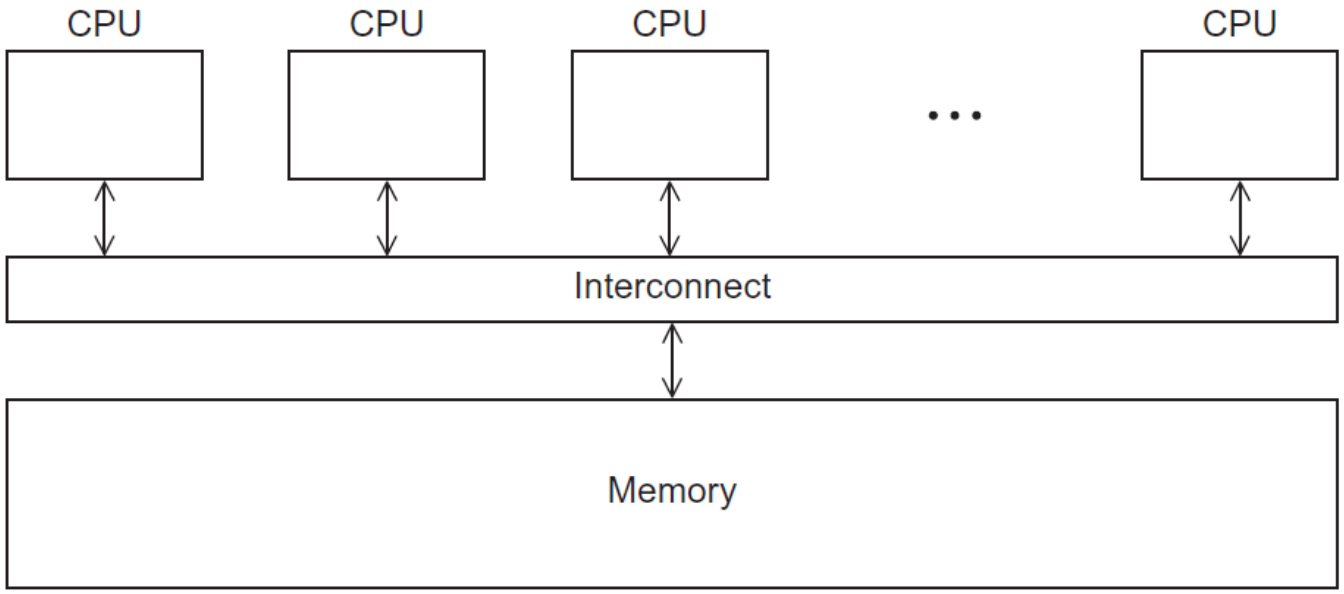
MIMD

- 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)



UMA Multicore Systems

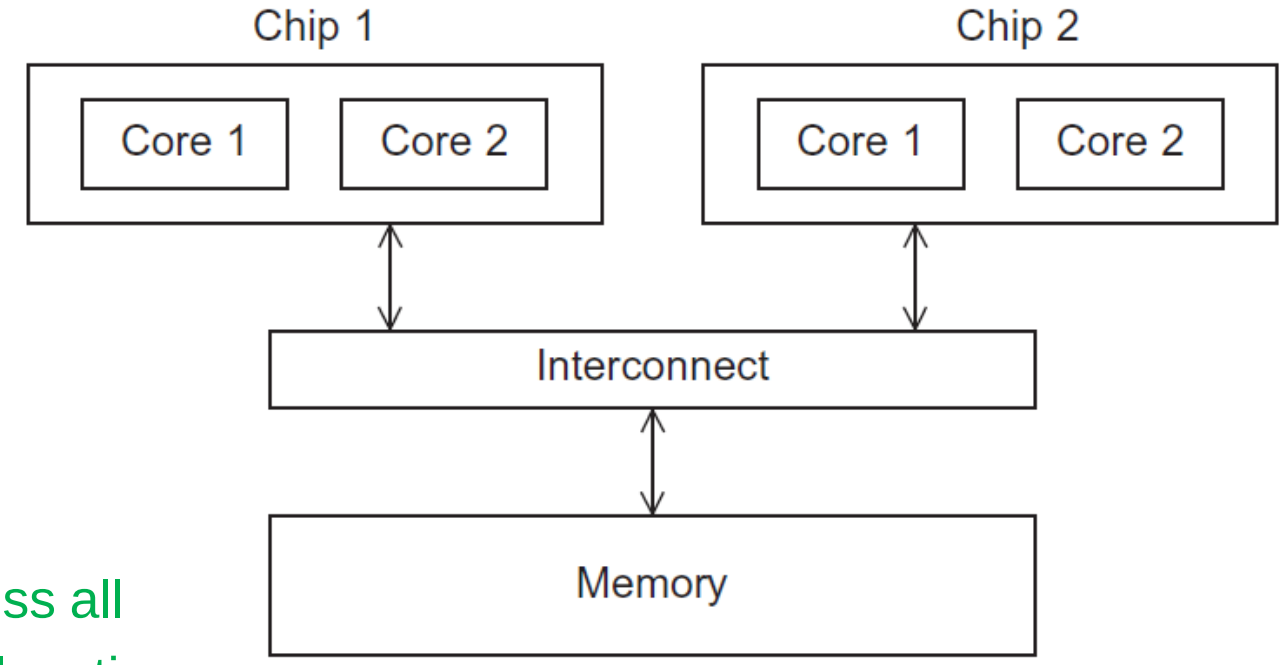
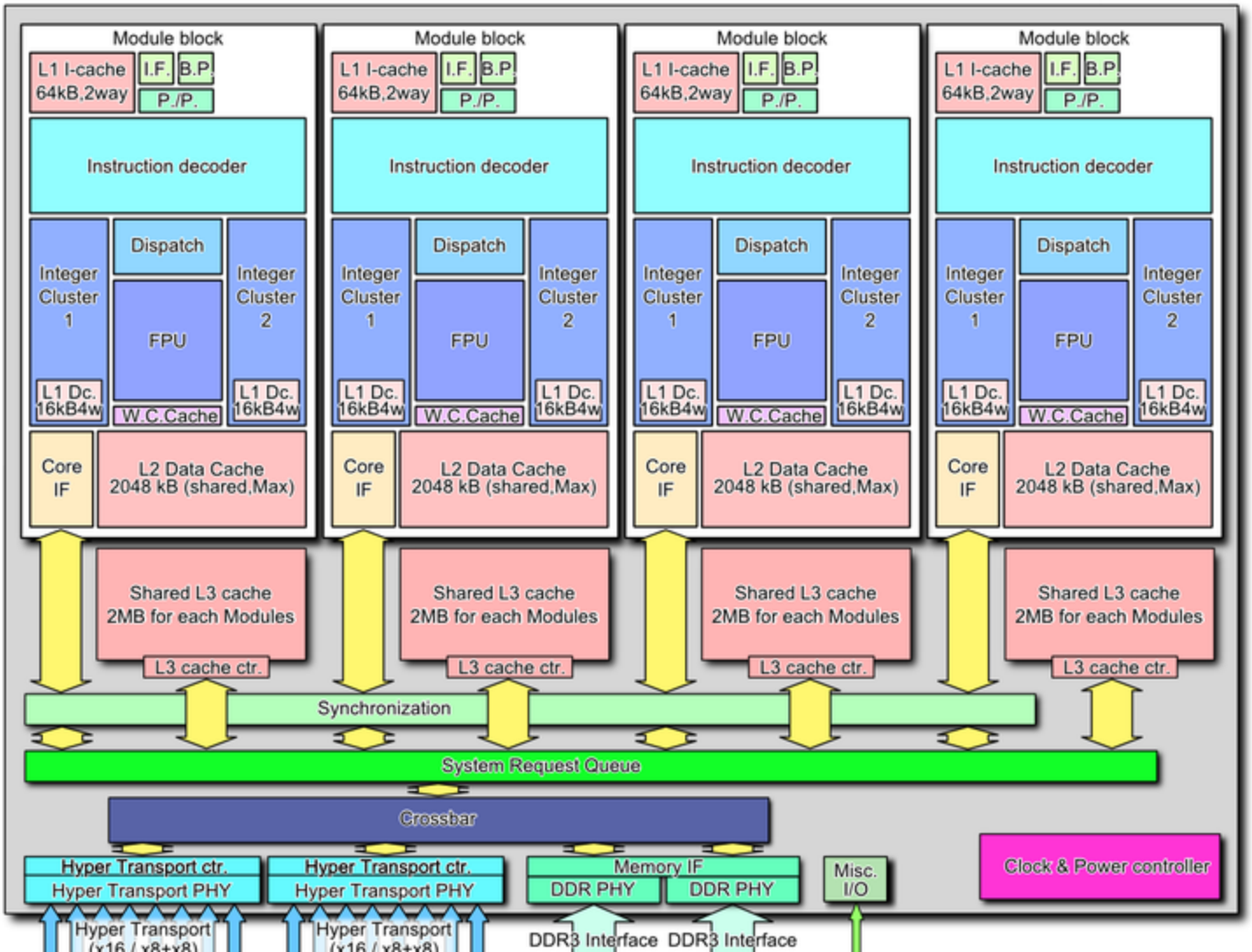


Figure 2.5

Time to access all the memory locations will be the same for all the cores.

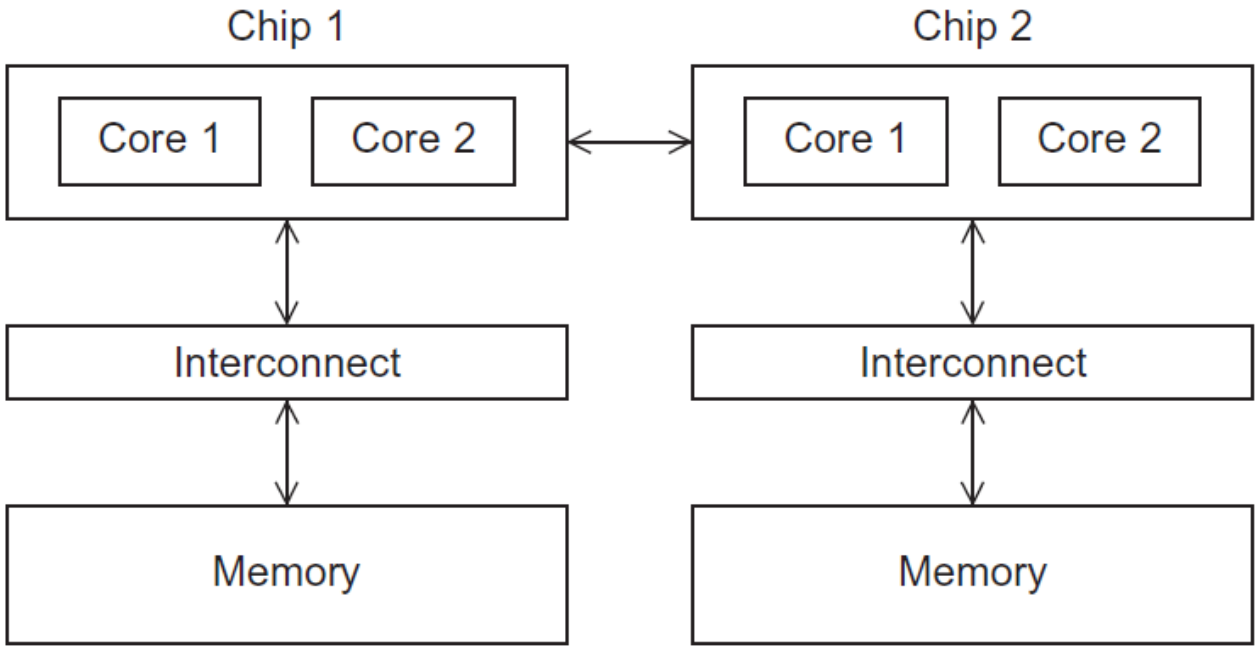
Copyright © 2010, Elsevier Inc. All rights Reserved

AMD 8-core CPU Bulldozer



sevier
erved

NUMA Multicore System



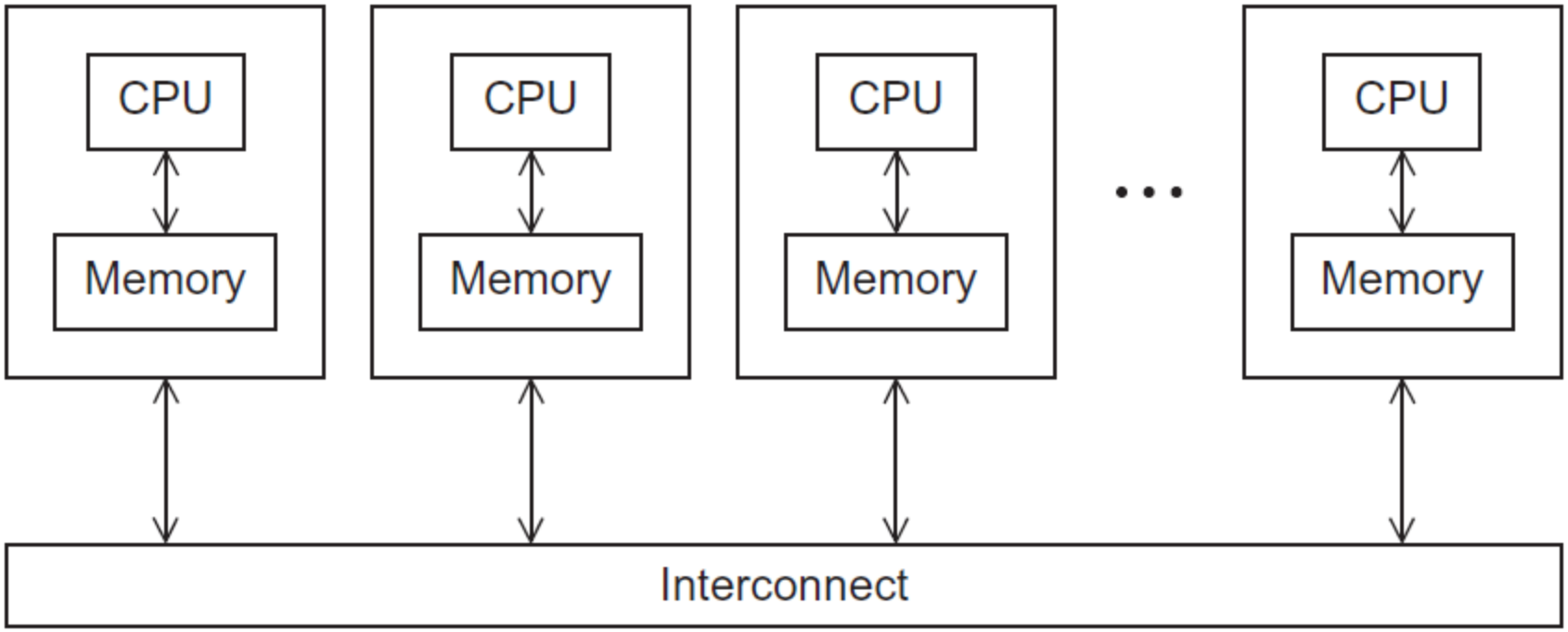
A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

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

Clustered Machines at a Lab and a Datacenter

Copyright © 2010, Elsevier Inc. All rights Reserved

Interconnection networks

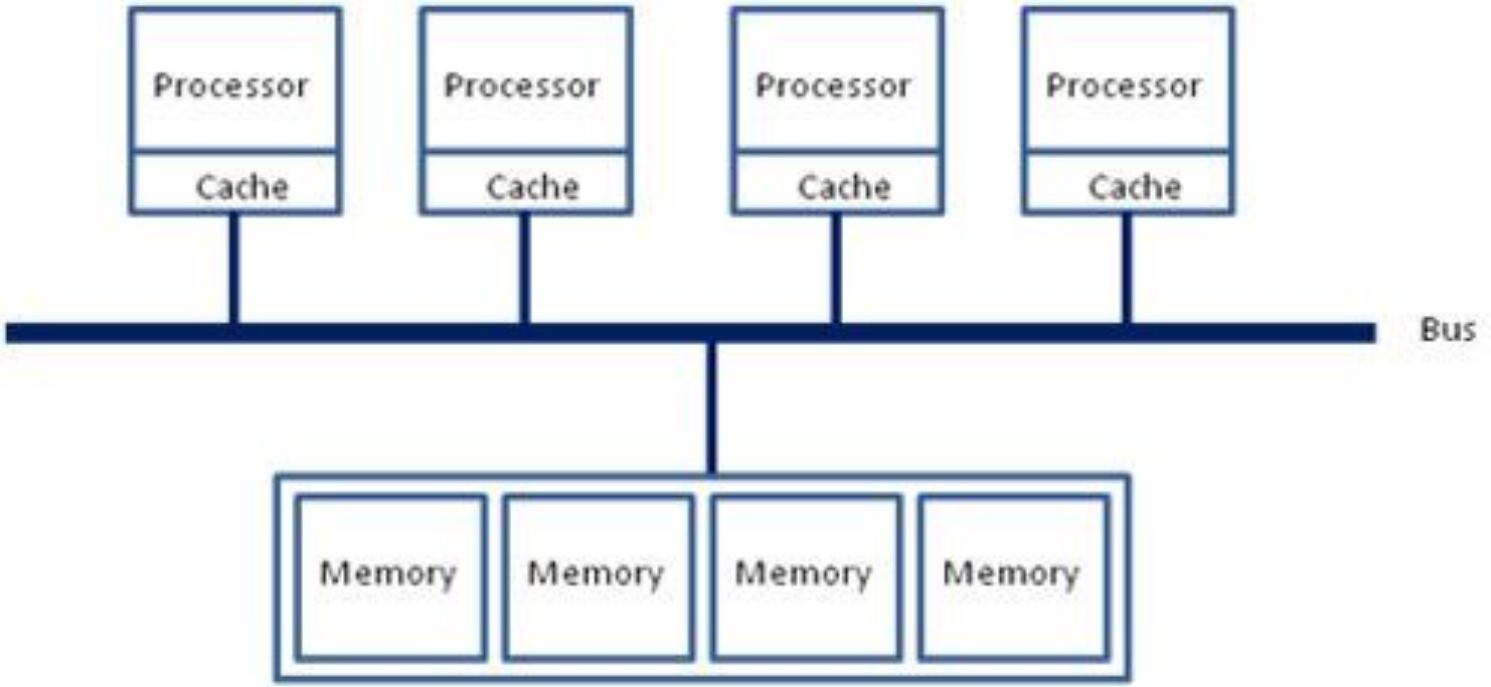
- **Affects performance of both distributed and shared memory systems.**

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

Copyright © 2010, Elsevier Inc. All rights Reserved

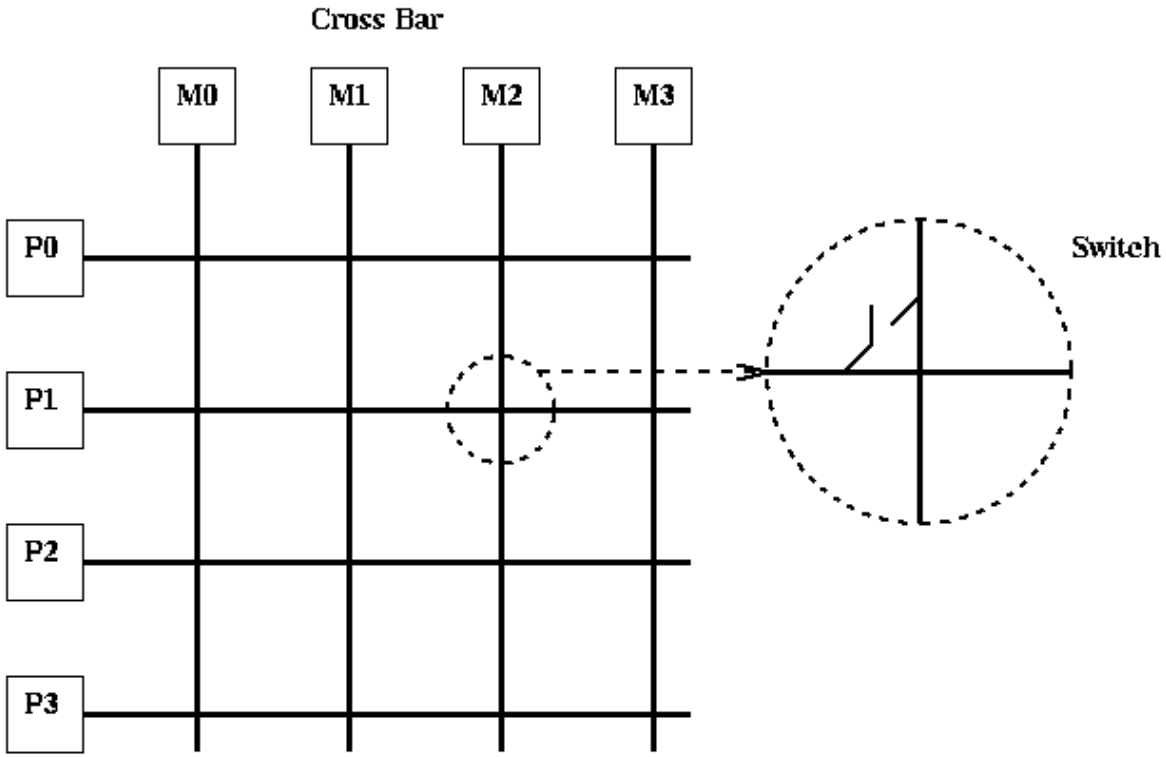
Shared memory interconnects: Bus

- 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.

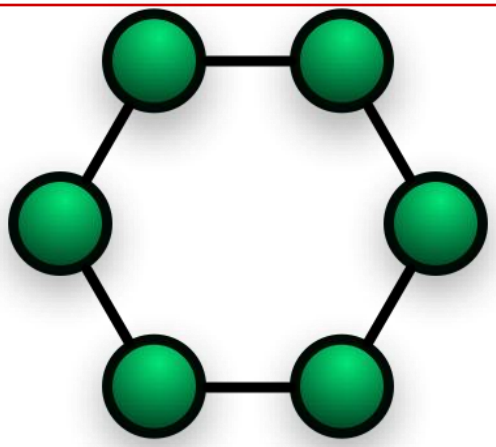


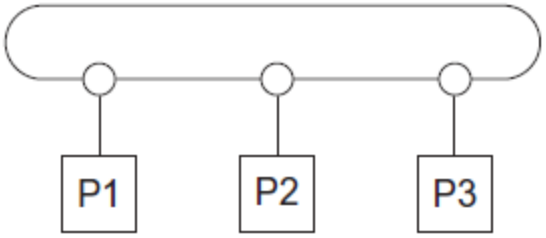
Distributed memory interconnects

- **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

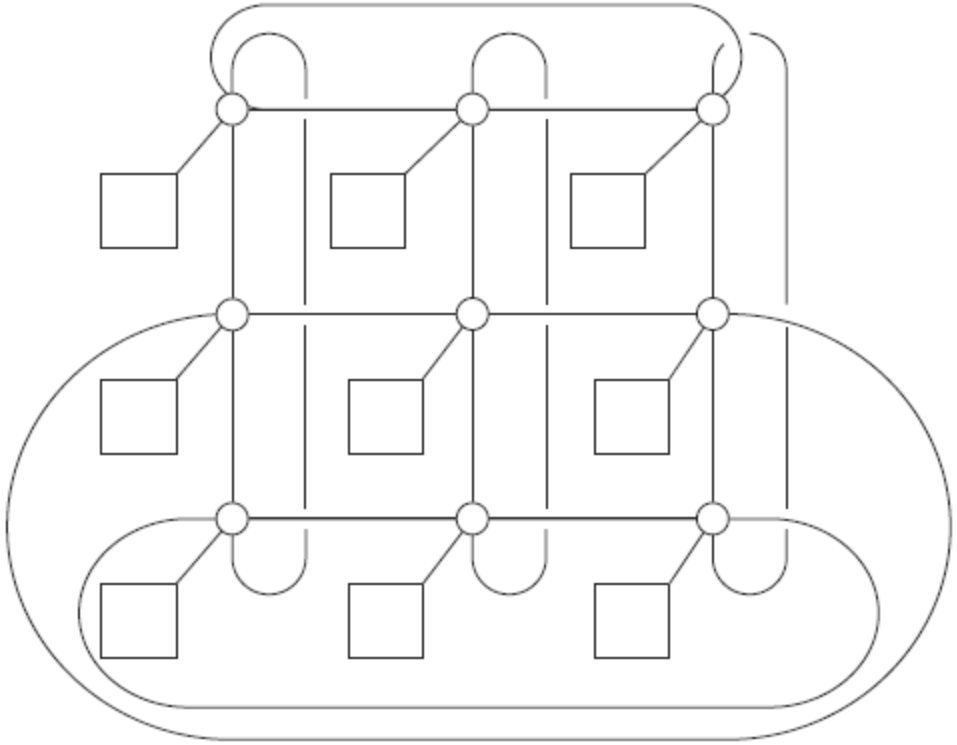
Direct interconnect





(a)

ring

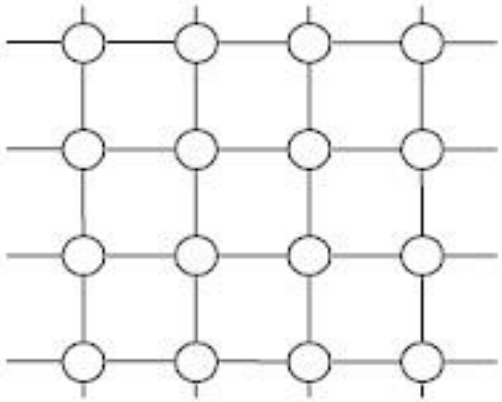


(b)

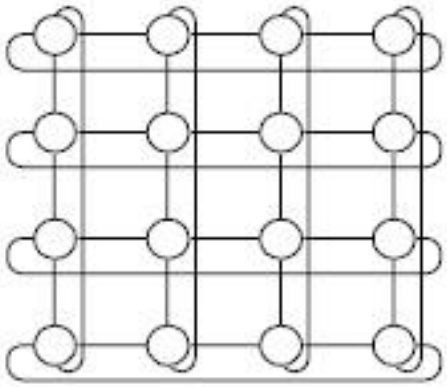
2D torus (toroidal mesh)

Copyright © 2010, Elsevier Inc. All rights Reserved

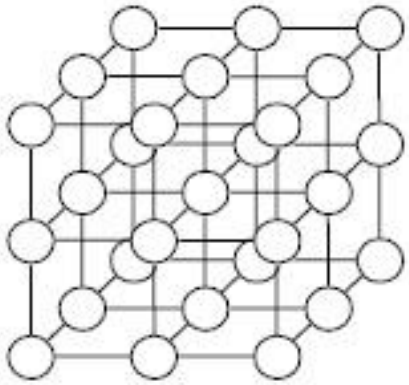
Direct interconnect: 2D Mesh vs 2D Torus



2D mesh



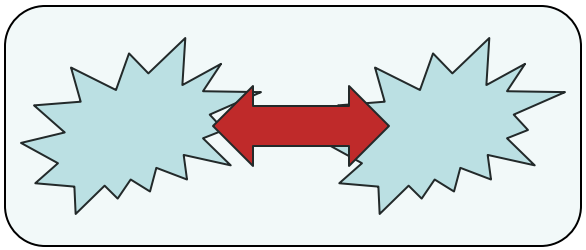
2D torus



3D mesh

How to measure network quality?

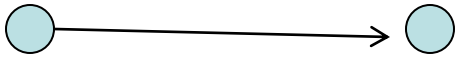
- **Bandwidth**
  - The rate at which a link can transmit data.
  - Usually given in megabits or megabytes per second.
- **Bisection width**
  - A measure of “number of simultaneous communications” between two subnetworks within a network



  - 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)

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.
  - Sometime it is called *startup cost.*

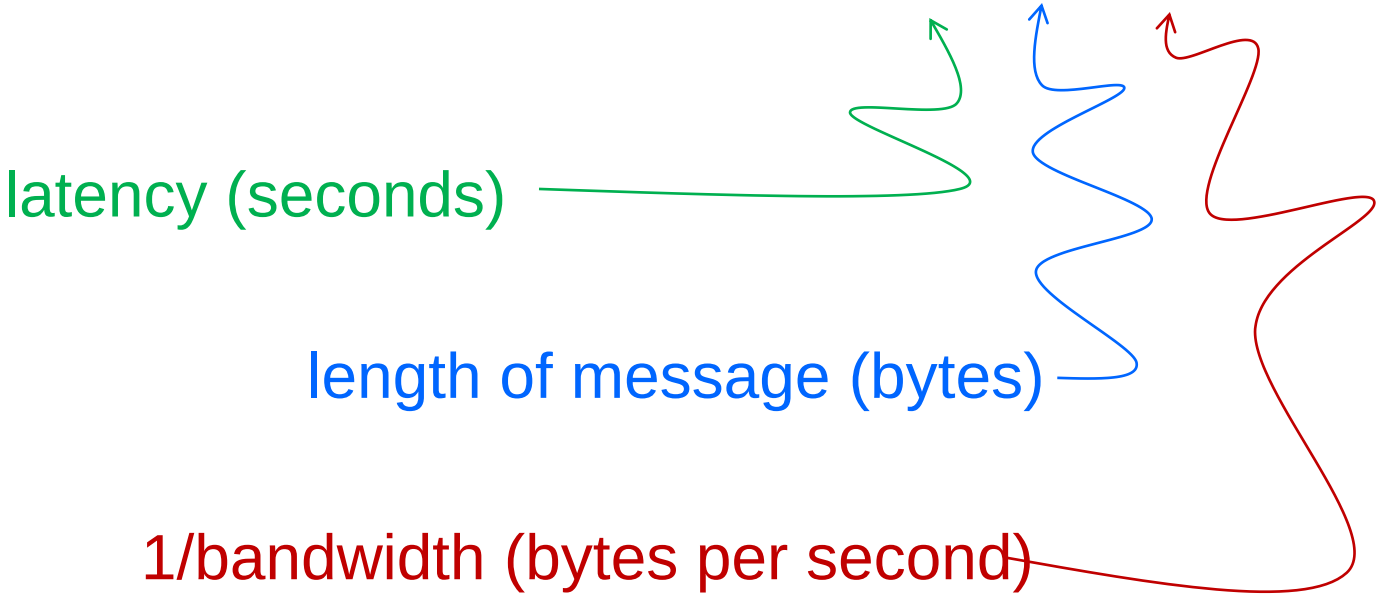


- **Bandwidth**
  - The rate at which the destination receives data after it has started to receive the first byte.

Copyright © 2010, Elsevier Inc. All rights Reserved

Network transmission cost

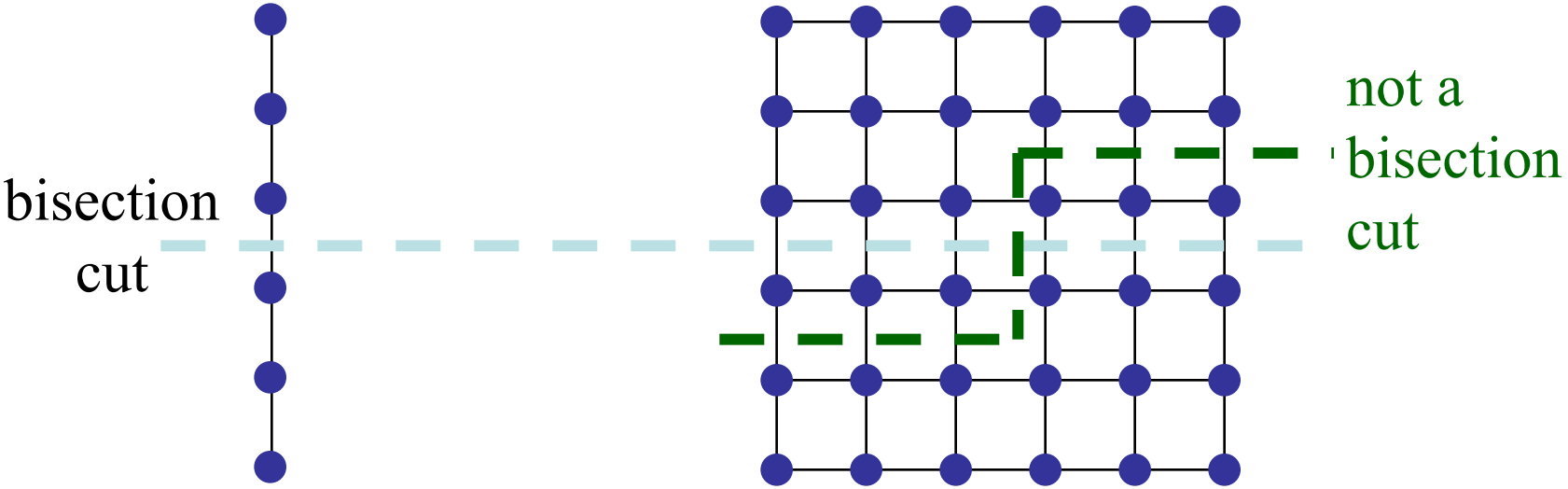
Message transmission time = $\alpha + m\,\beta$



Typical latency/startup cost: 2 microseconds ~ 1 millisecond
Typical bandwidth: 100 MB ~ 1GB per second

Bisection width vs Bisection bandwidth

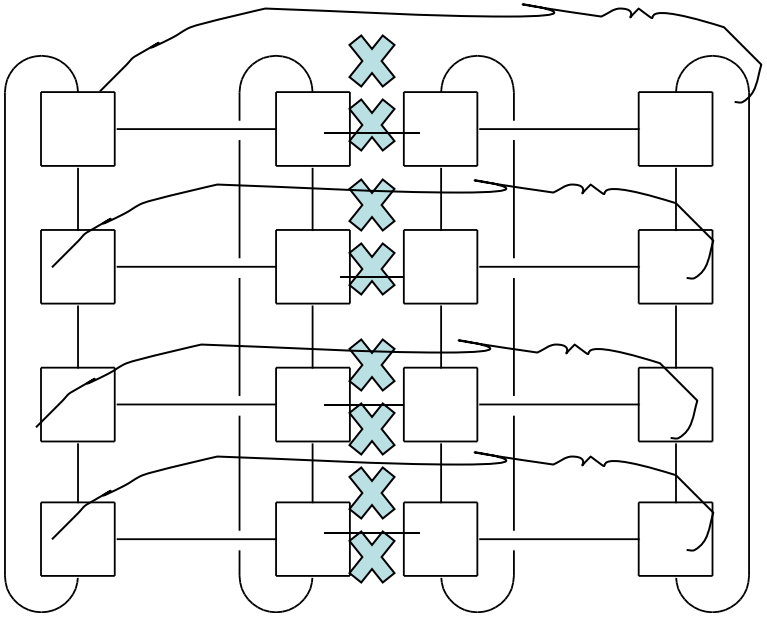
- **Example of bisection width**

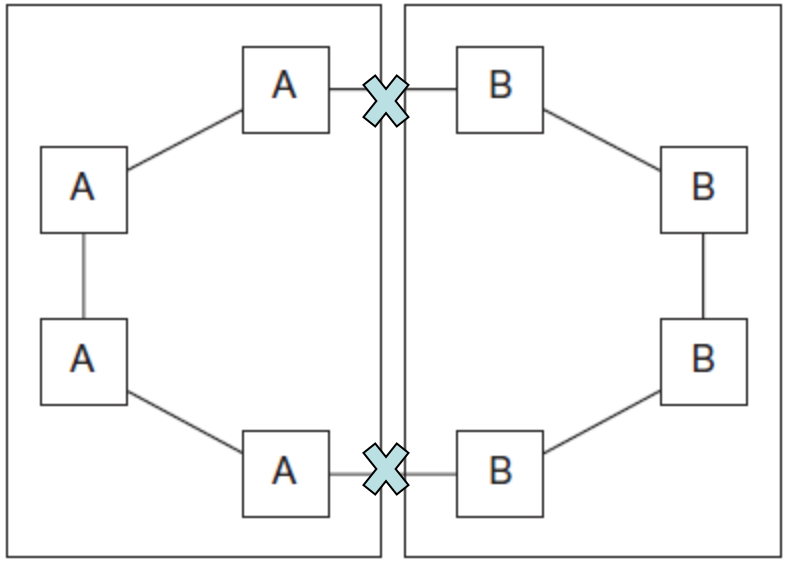


- **Bisection bandwidth**
  - Sum bandwidth of links that cut the network into two equal halves.
  - Choose the minimum one.

Bisection width: more examples

A bisection of a 2D torus with p nodes: ? 2 sqrt(p)





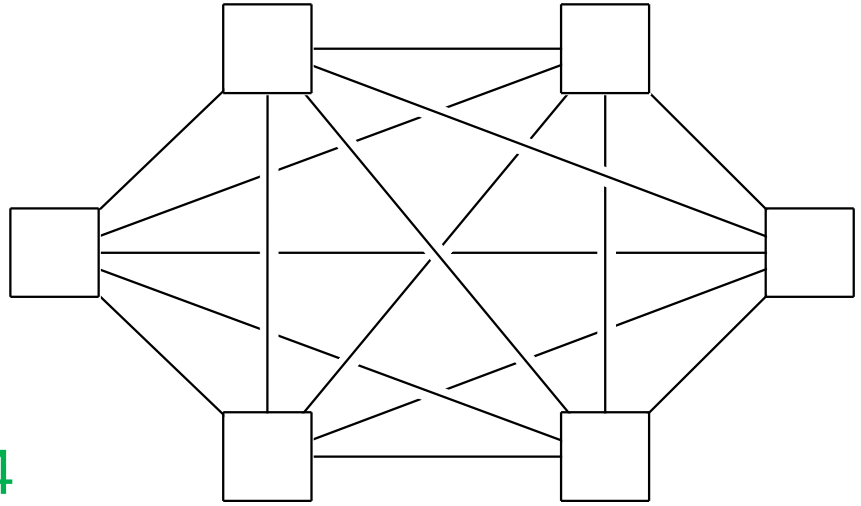
(a)

Copyright © 2010, Elsevier Inc. All rights Reserved

Fully connected network

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

impractical



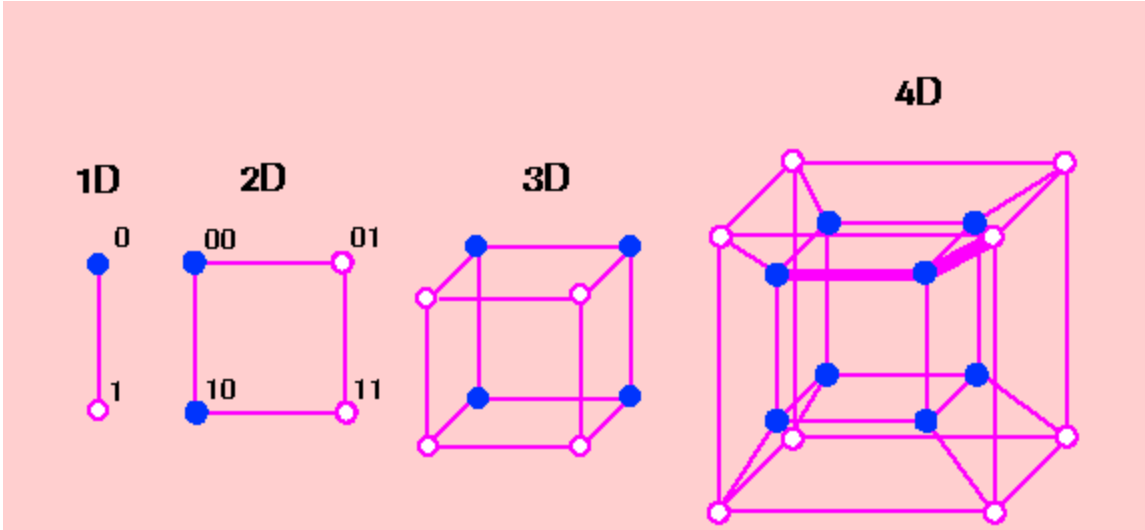
bisection width = $p^2/4$

Figure 2.11

Copyright © 2010, Elsevier Inc. All rights Reserved

Hypercube

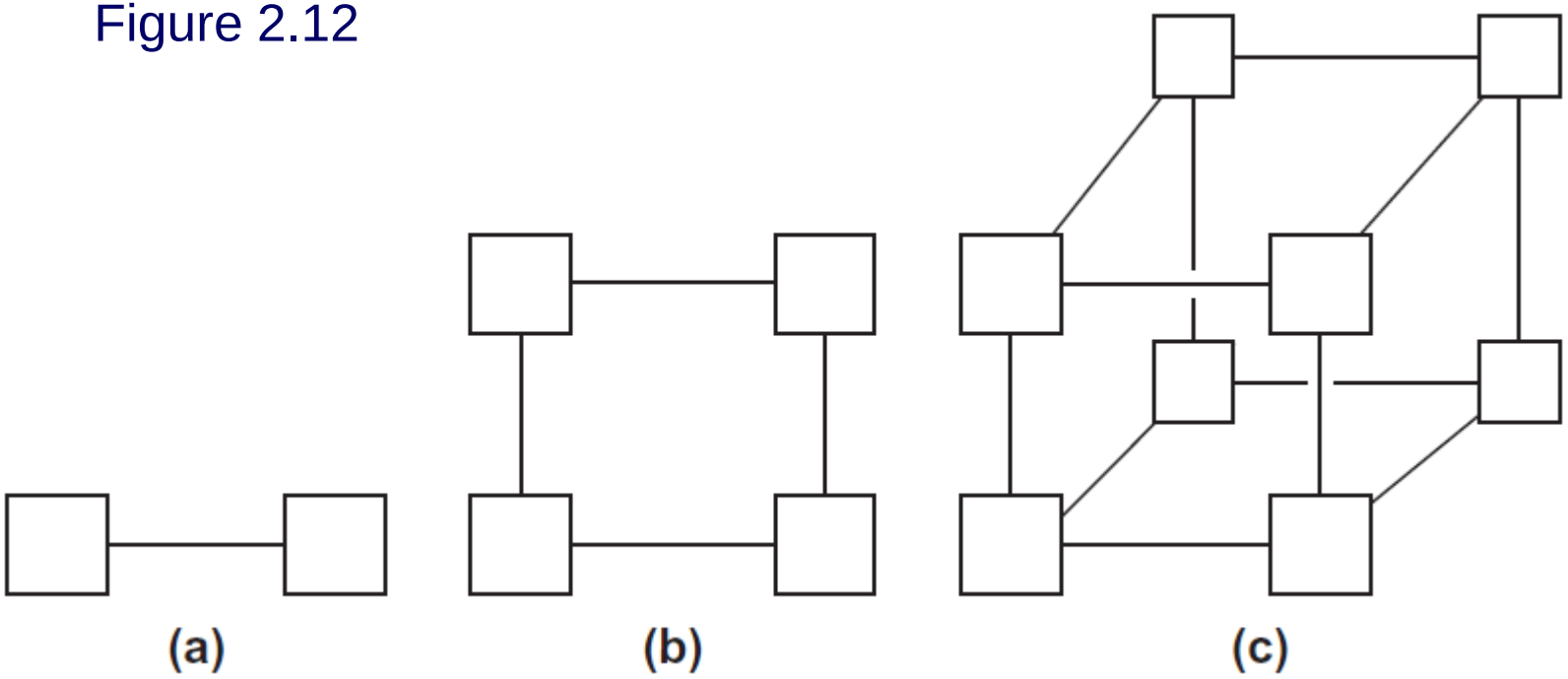
- **Built inductively:**
  - A one-dimensional hypercube is a fully-connected system with two processors.
  - A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches.
  - Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.



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

Hypercubes

Figure 2.12



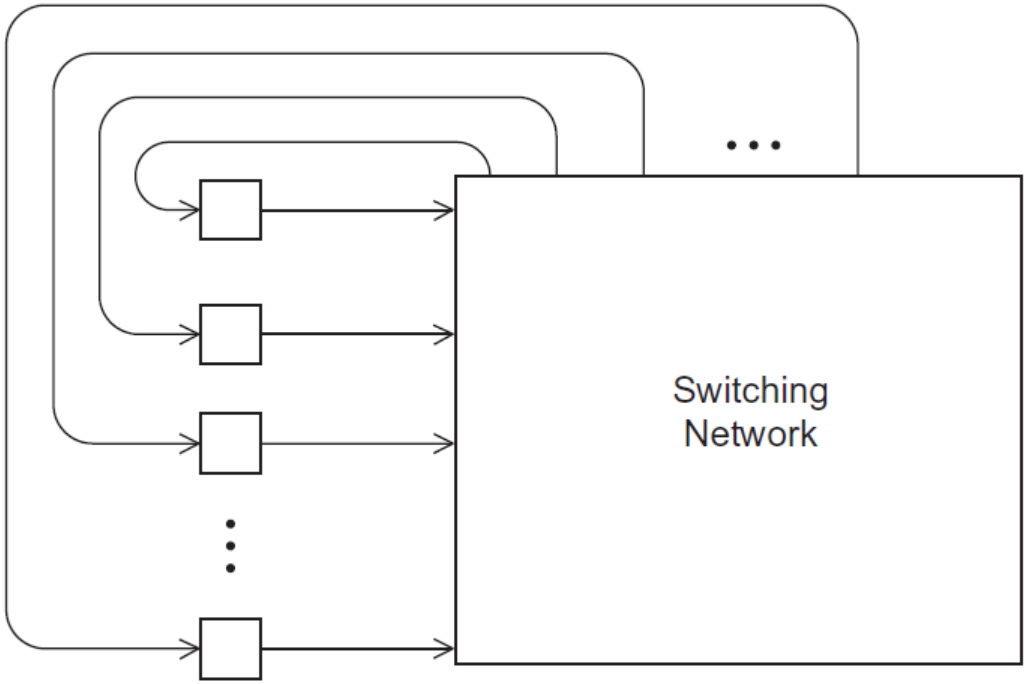
one- two- three-dimensional

Copyright © 2010, Elsevier Inc. All rights Reserved

Indirect interconnects

- **Simple examples of indirect networks:**
  - Crossbar
  - Omega network

- **A generic indirect network**



.0, Elsevier
; Reserved

Crossbar indrect interconnect

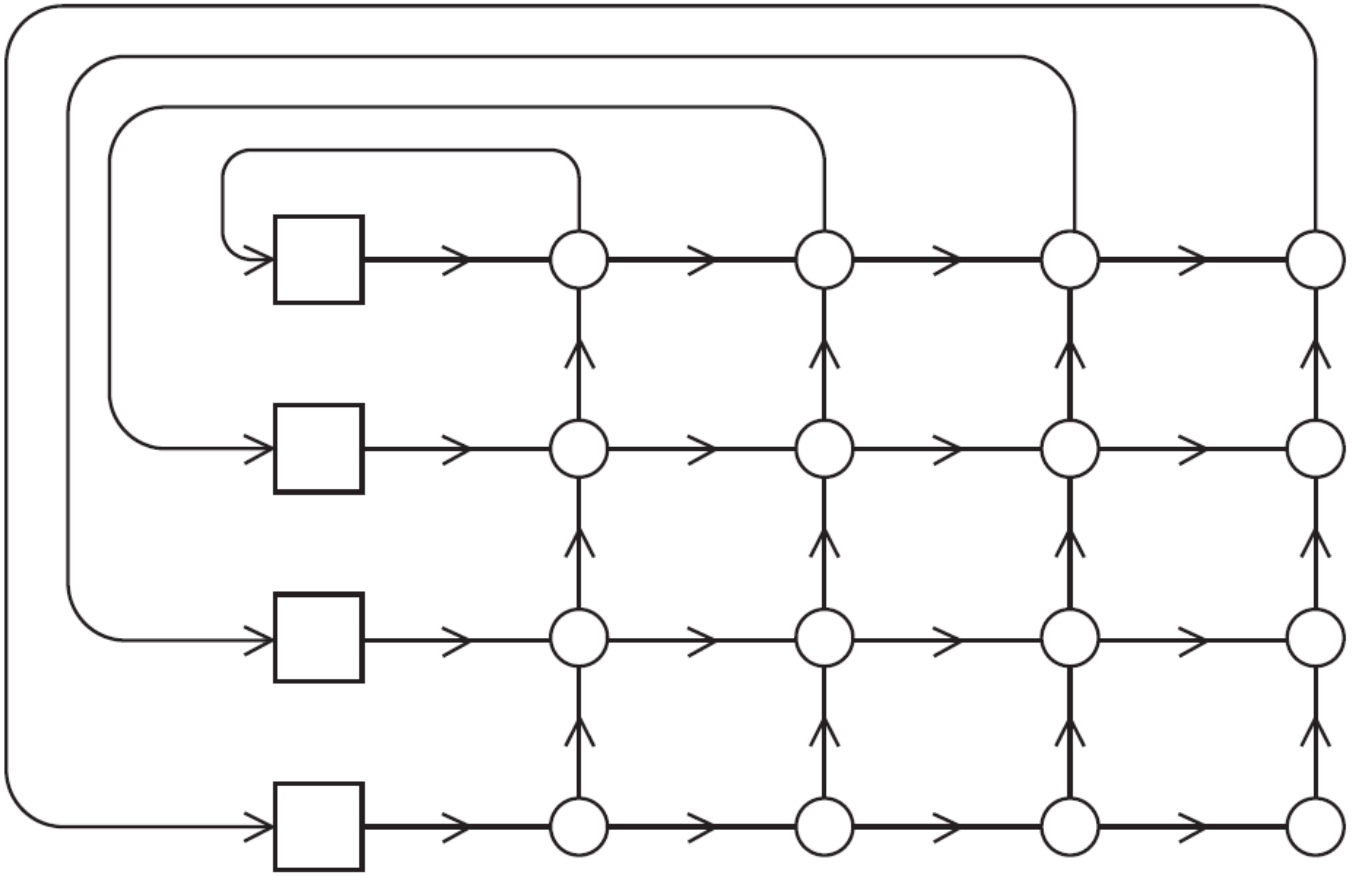


Figure 2.14

Copyright © 2010, Elsevier Inc. All rights Reserved

An omega network

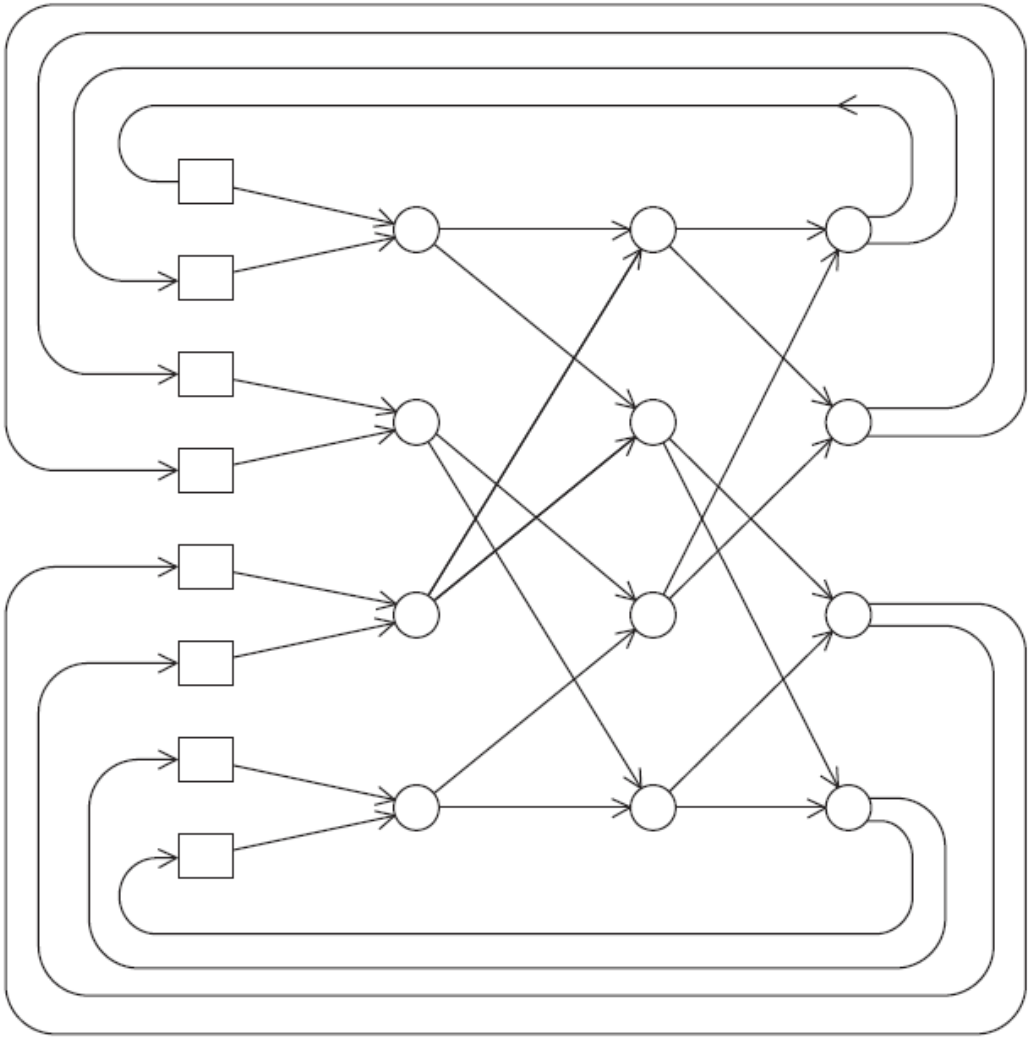
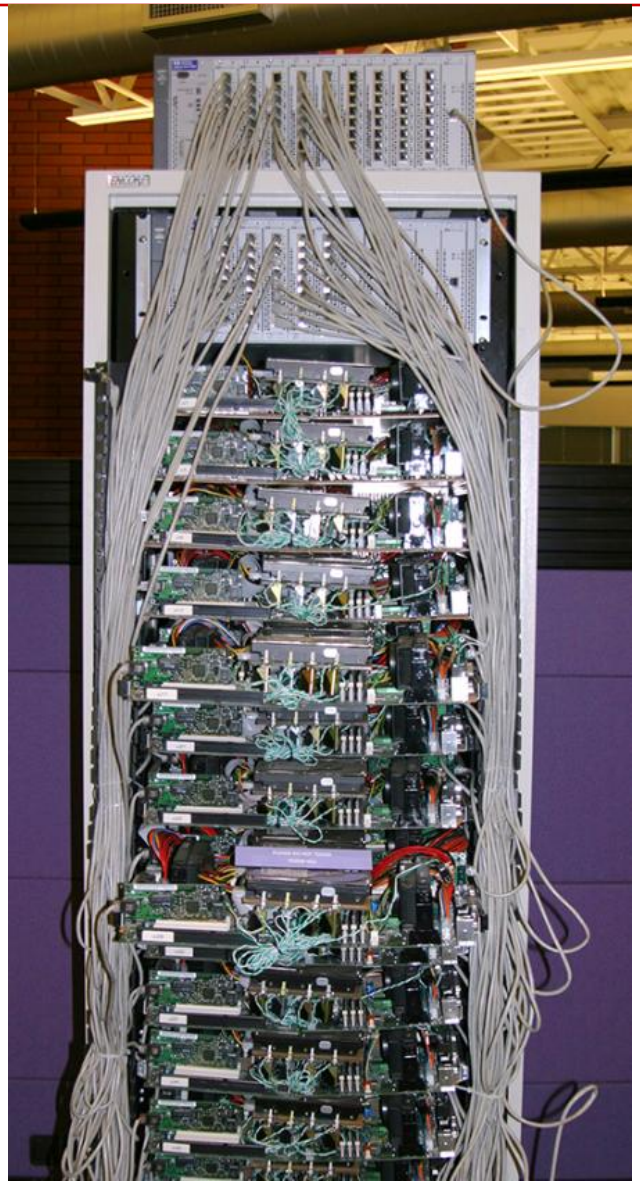


Figure 2.15

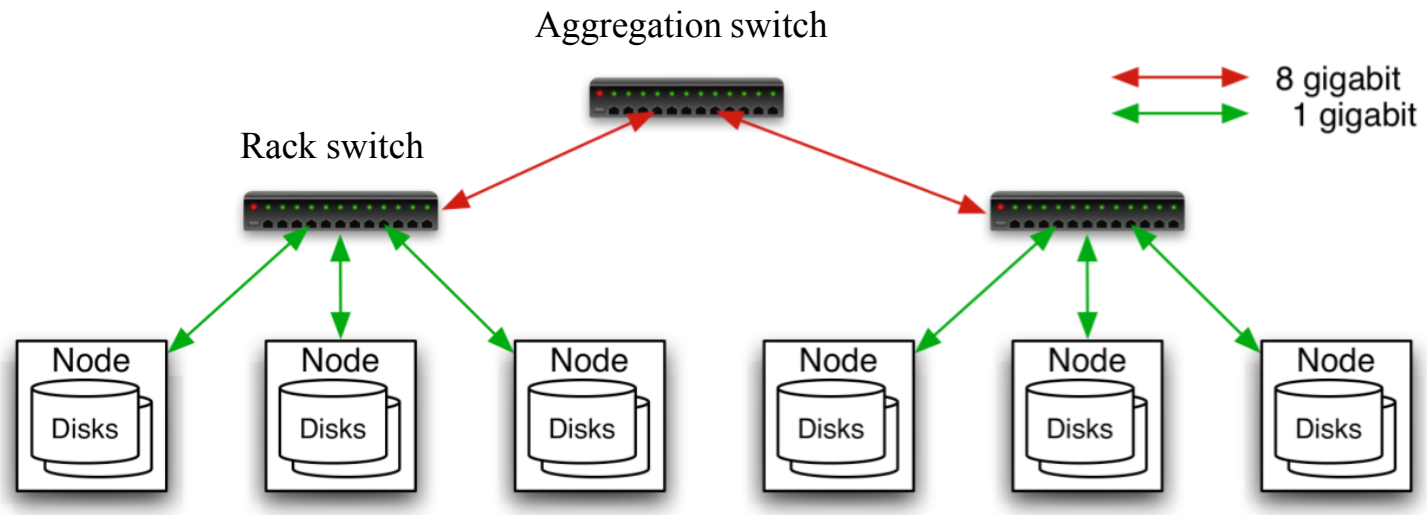
2010, Elsevier
ghts Reserved

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**

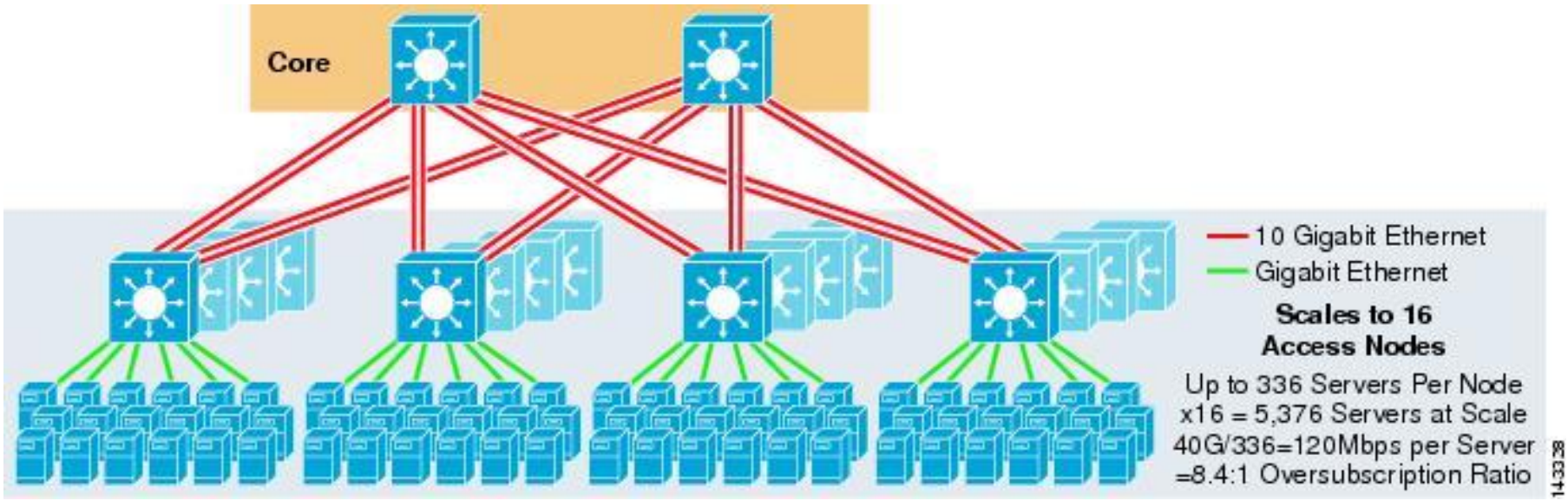


Typical network for a cluster



- **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

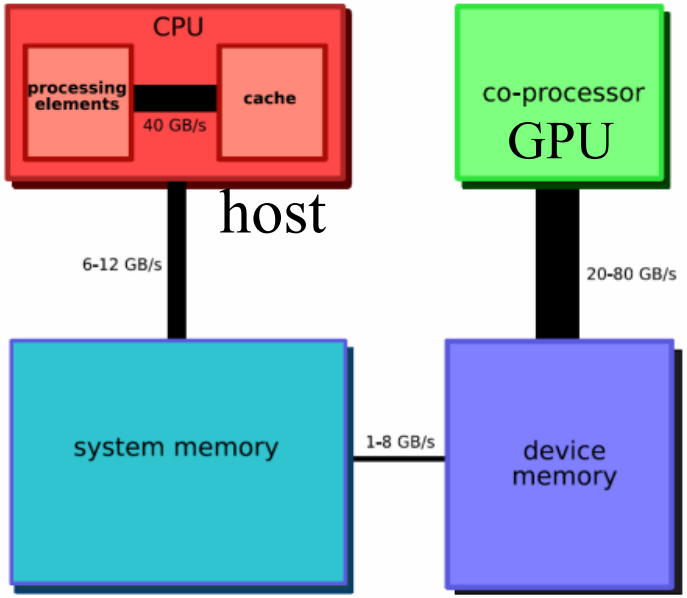
Layered Network in Clustered Machines

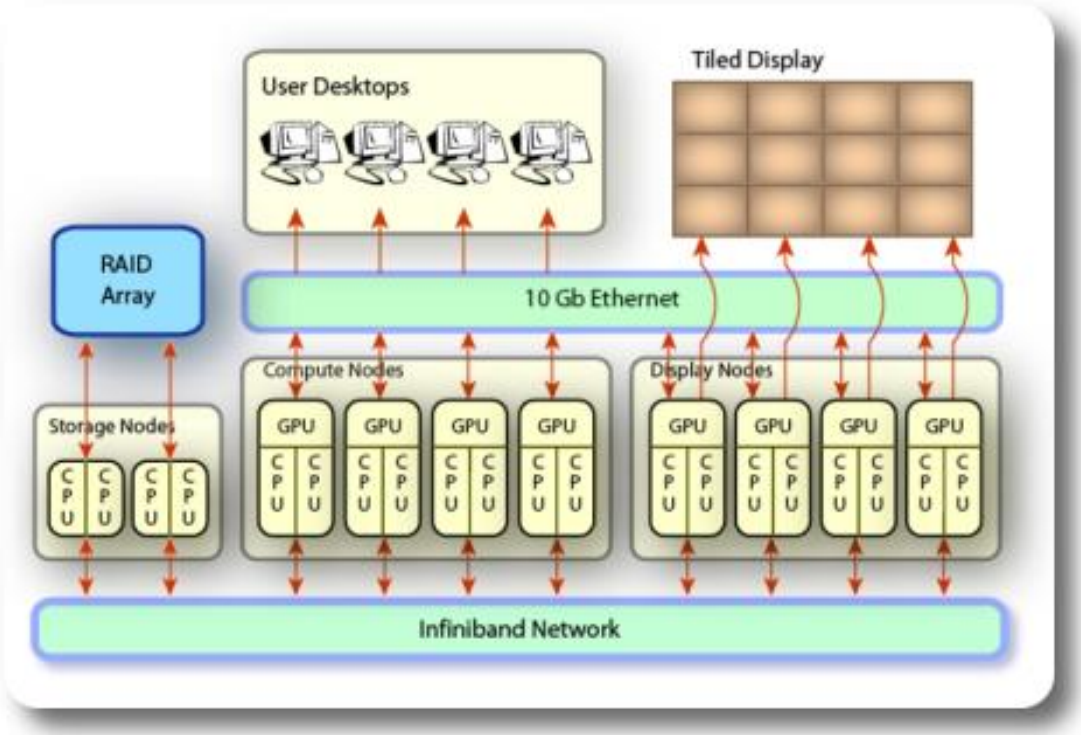


- **A layered example from Cisco: core, aggregation, the edge or top-of-rack switch.**
- **http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_3a.html**

Hybrid Clusters with GPU

Node in a CPU/GPU cluster

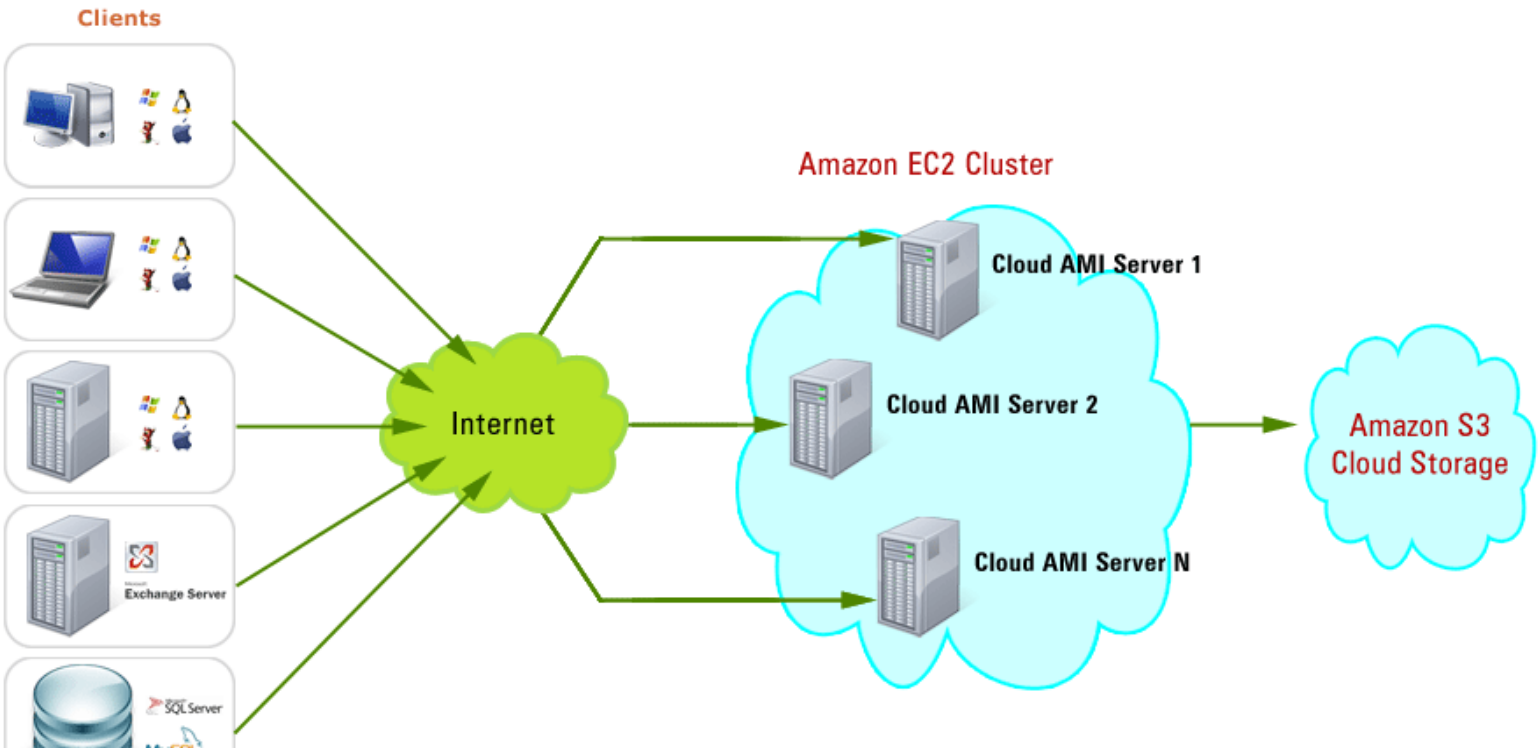




- A Maryland cluster couples CPUs, GPUs, displays, and storage.
- Applications in visual and scientific computing

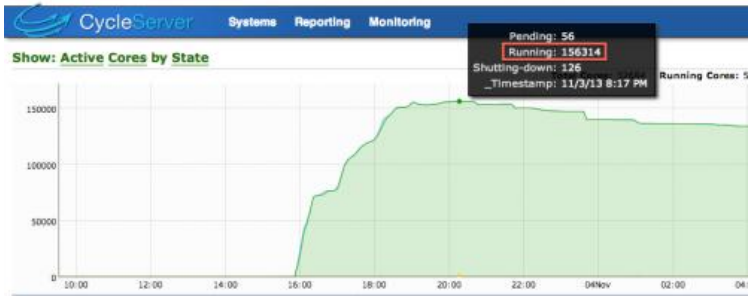
Cloud Computing with Amazon EC2

- **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



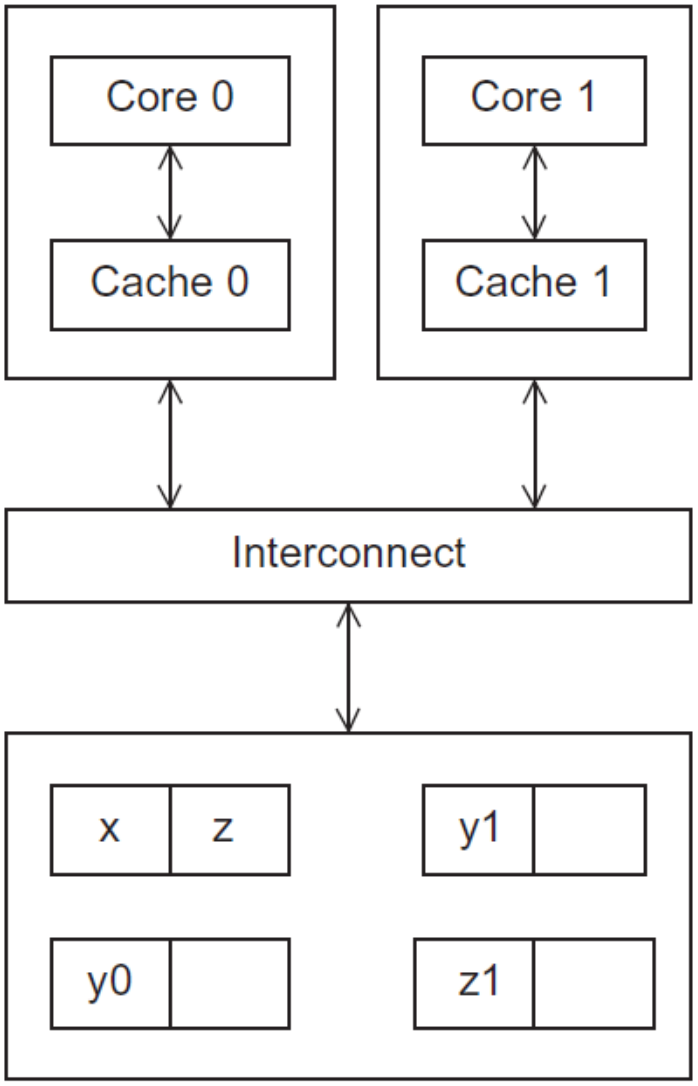
Usage Examples with Amazon EC2

- **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 example: 82*8*52=$34,112
  - **Otherwise:** ~$150+K to purchase + annual datacenter cost.
- **Another example from Cycle Computing Inc**
  - Run 205,000 molecule simulation with 156,000 Amazon cores for 18 hours -- $33,000.



Cache coherence

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



Cache coherence issue

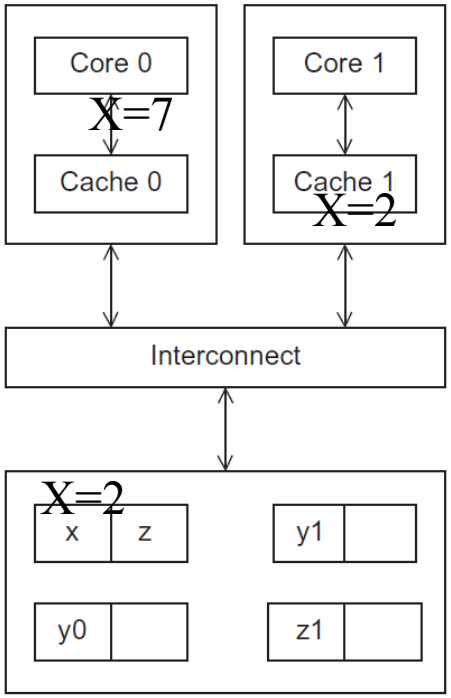
x = 2; /* shared variable */

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

y0 eventually ends up = 2

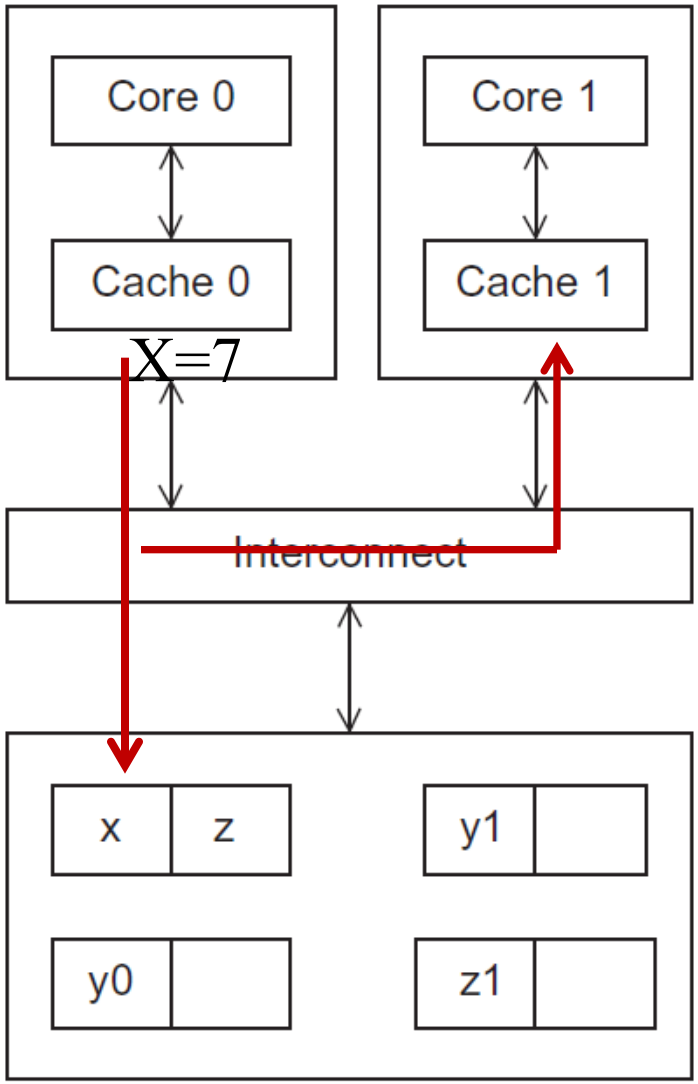
y1 eventually ends up = 6

z1 = 4*7 or 4*2?



Snooping Cache Coherence

- All cores share a bus .
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.



PARALLEL SOFTWARE

Copyright © 2010, Elsevier Inc. All rights Reserved

The burden is on software

- **Hardware and compilers can keep up the pace needed.**
- **From now on…**
  - In shared memory programs:
    - Start a single process and fork threads.
    - Threads carry out tasks.
  - In distributed memory programs:
    - Start multiple processes.
    - Processes carry out tasks.

Copyright © 2010, Elsevier Inc. All rights Reserved

SPMD – single program multiple data

- **A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.**

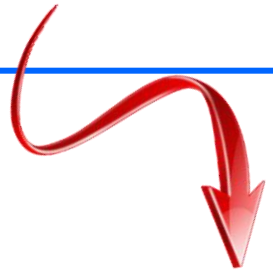
```
if (I'm thread/process i)
    do this;
else
    do that;
```

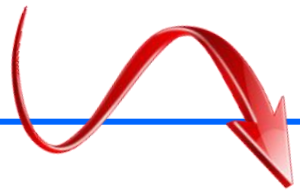
- **Shared memory machines → threads (or processes)**
- **Distributed memory machines → processes**

Copyright © 2010, Elsevier Inc. All rights Reserved

Challenges: Nondeterminism of execution order

```
. . .
printf ( "Thread %d > my_val = %d\n" ,
         my_rank , my_x ) ;
. . .
```





Execution 1:

Thread 1 > my_val = 19

Thread 0 > my_val = 7

Execution 2:

Thread 0 > my_val = 7

Thread 1 > my_val = 19

Copyright © 2010, Elsevier Inc. All rights Reserved

Input and Output

- **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);

PERFORMANCE

Copyright © 2010, Elsevier Inc. All rights Reserved

Speedup

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

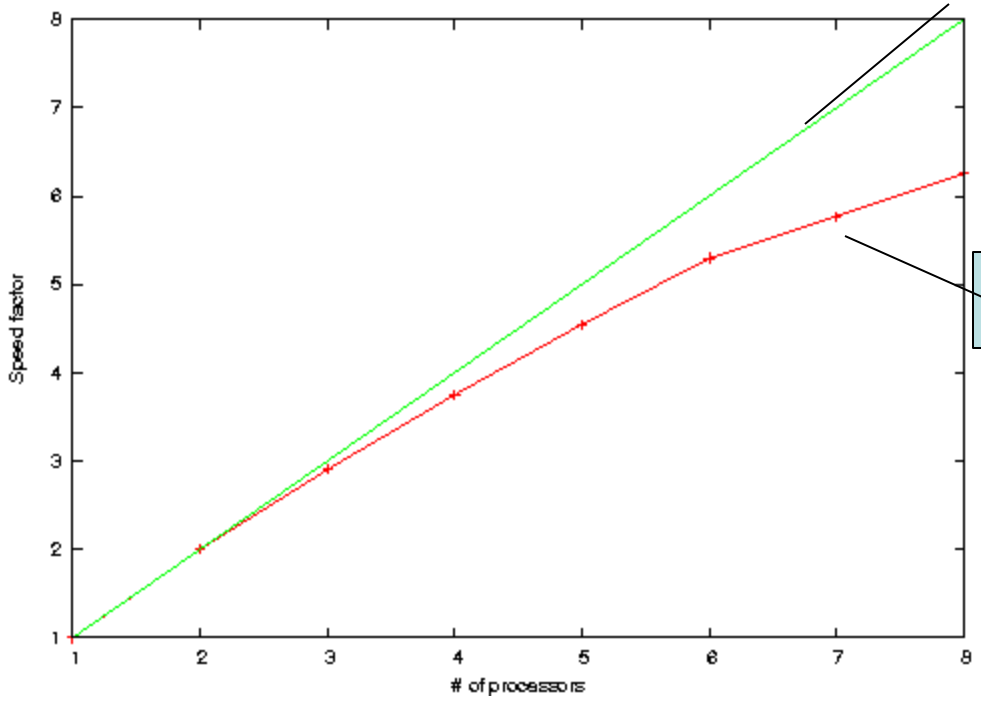
If linear speedup

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Speedup of a parallel program

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



Perfect speedup

Actual speedup

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

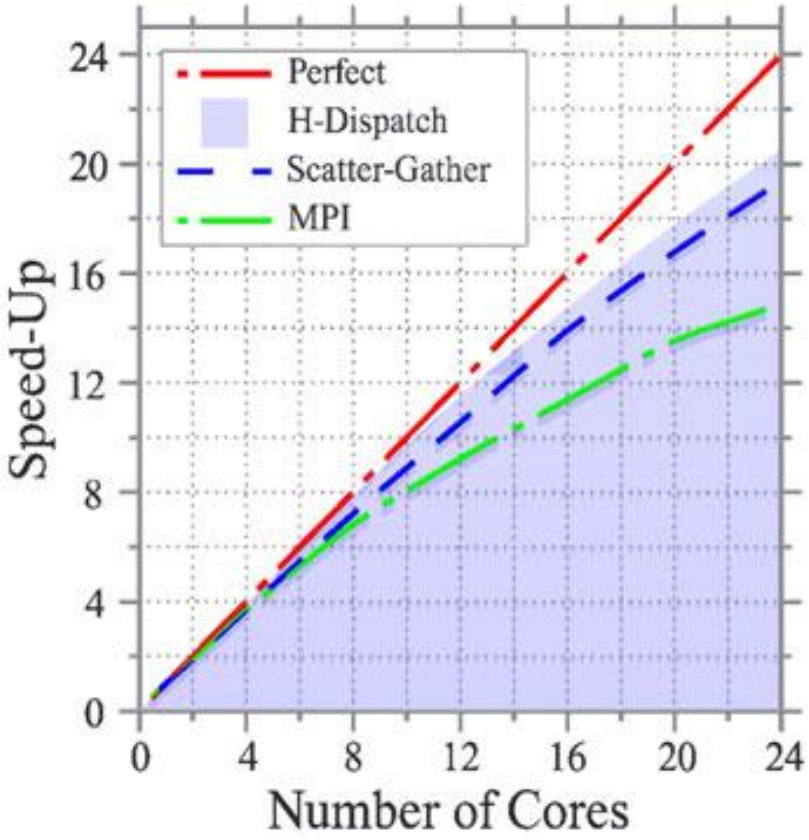
$$E = \frac{\text{Speedup}}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p\, T_{parallel}}$$

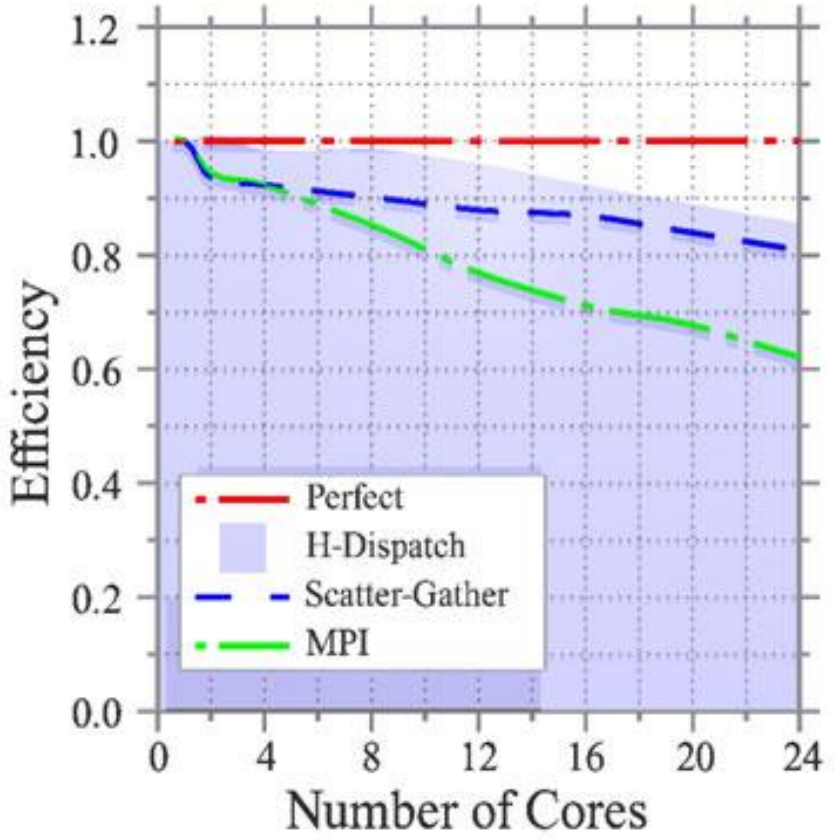
Measure how well-utilized the processors are, compared to effort wasted in communication and synchronization.

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



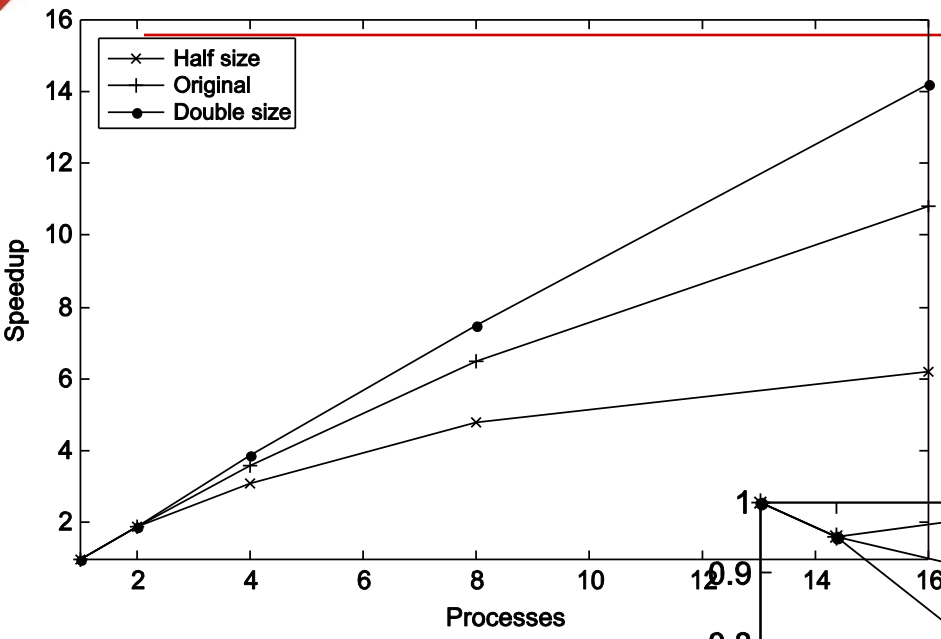


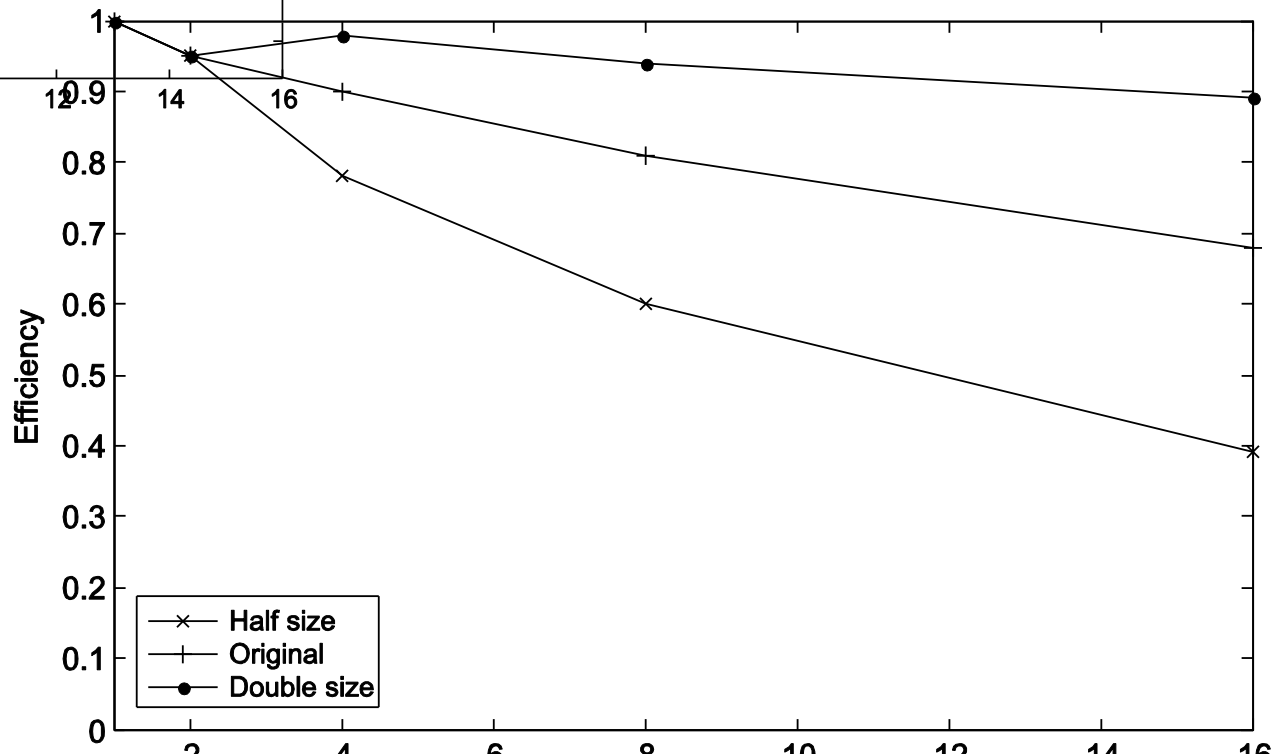
Impact of Problem Sizes on Speedups and efficiencies

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Copyright © 2010, Elsevier Inc. All rights Reserved

Problem Size Impact on Speedup and Efficiency





Strongly scalable

- If we increase the number of processors (processes/threads) , efficiency is fixed without increasing problem size. This solution is *strongly scalable*.
  - Ex.
    - Seq = n^2 PT = $(n+n^2)/p$
    - Efficiency = $n^2/(n^2+n)$
  - Not strongly scalable:
    - PT = $n+n^2/p$
    - Efficiency = $n^2/(n^2+np)$

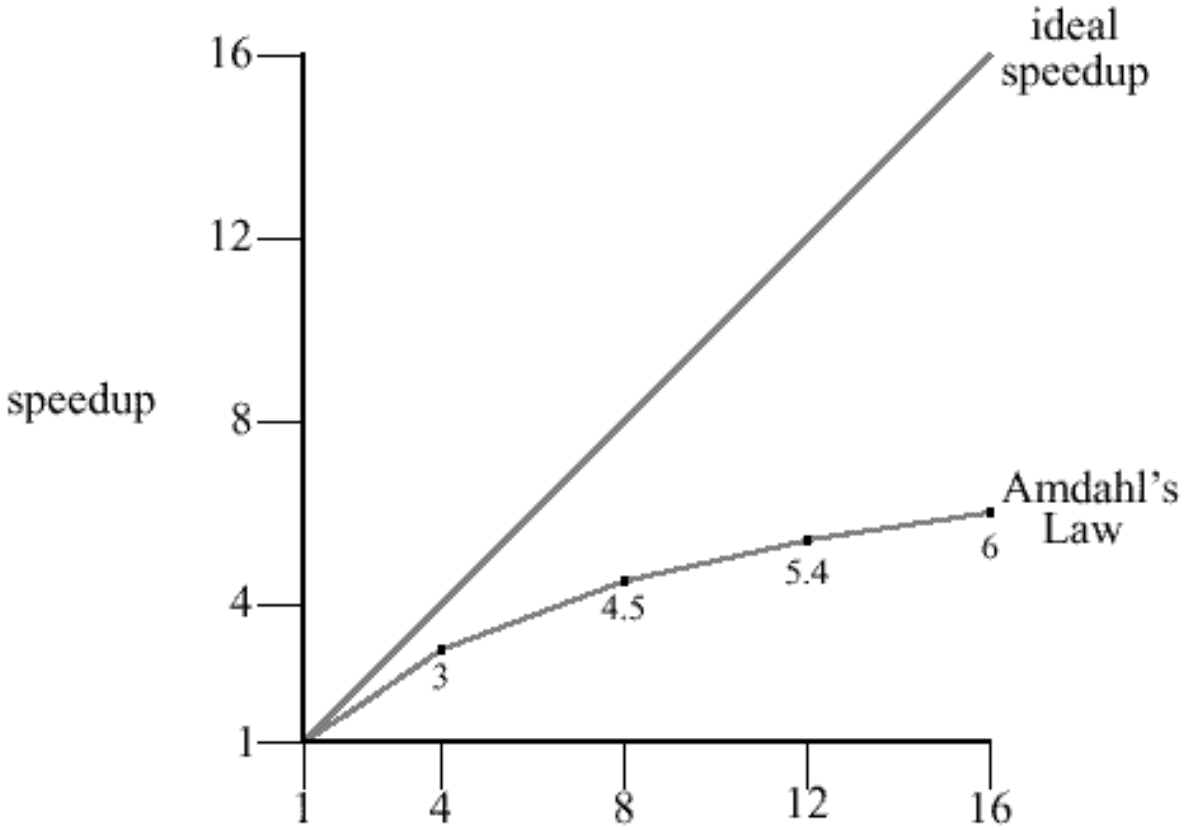
Copyright © 2010, Elsevier Inc. All rights Reserved

Weak Scalable

- Efficiency is fixed when increasing the problem size at the same rate as increasing the number of processors, the solution is *weakly scalable*.
  - Ex. Seq = n^2 PT = $n+n^2/p$
  - Efficiency = $n^2/(n^2 +np)$
  - Increase problem size and #processors by k
  - $(kn)^2/((kn)^2 +knkp) = n^2/(n^2 +np)$

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

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

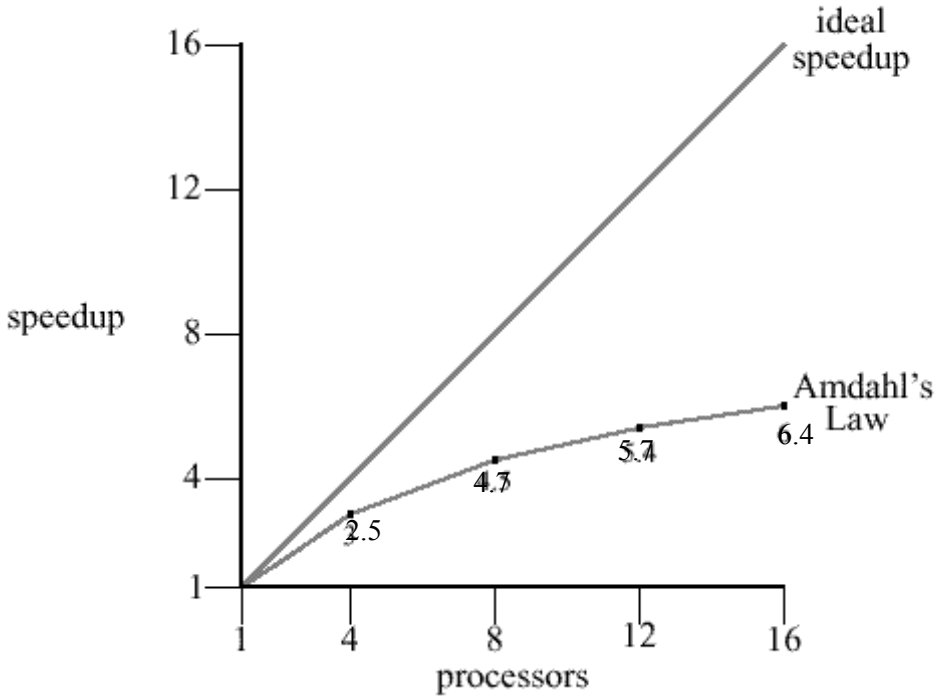
- $T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$

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);
```

Example function

MPI_Wtime() in MPI

gettimeofday() in Linux

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

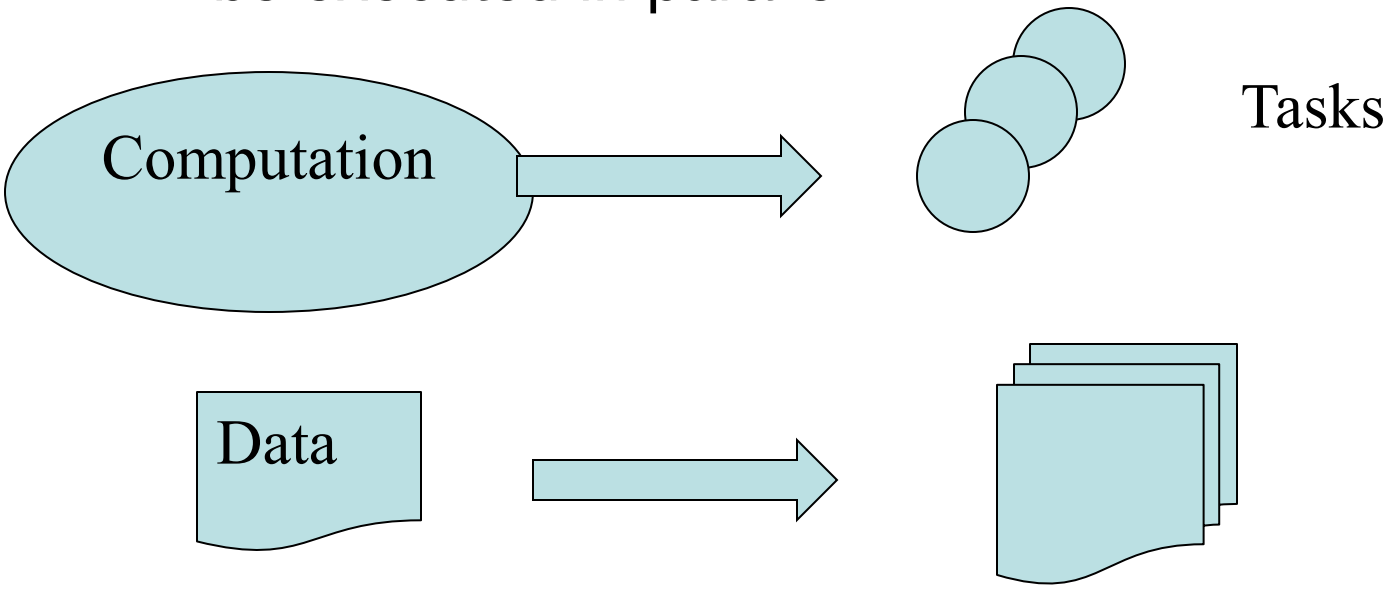
PARALLEL PROGRAM DESIGN

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.

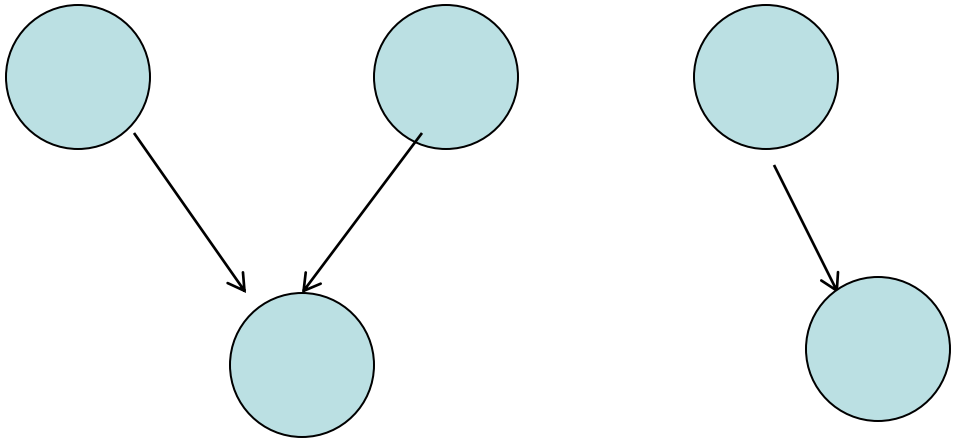
   The focus here should be on identifying tasks that can be executed in parallel.



Copyright © 2010, Elsevier Inc. All rights Reserved

Foster's methodology

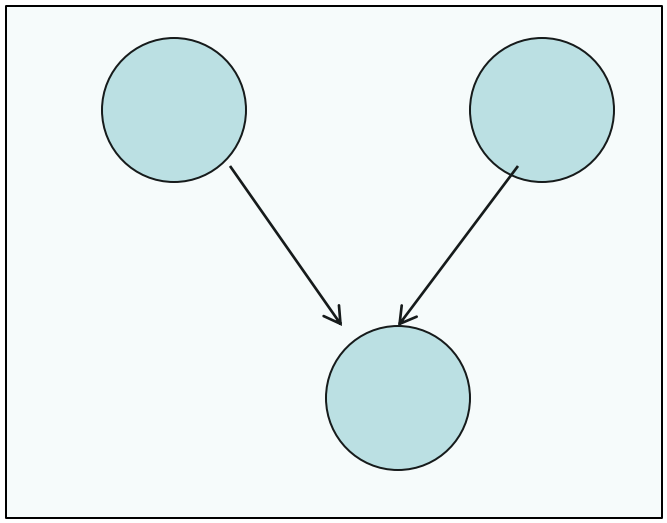
2. Communication:
   - Identify dependence among tasks
   - Determine inter-task communication

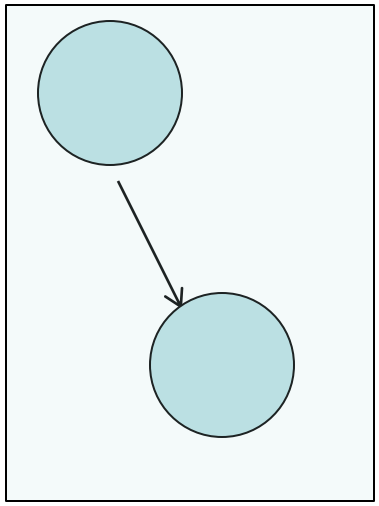


Copyright © 2010, Elsevier Inc. All rights Reserved

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



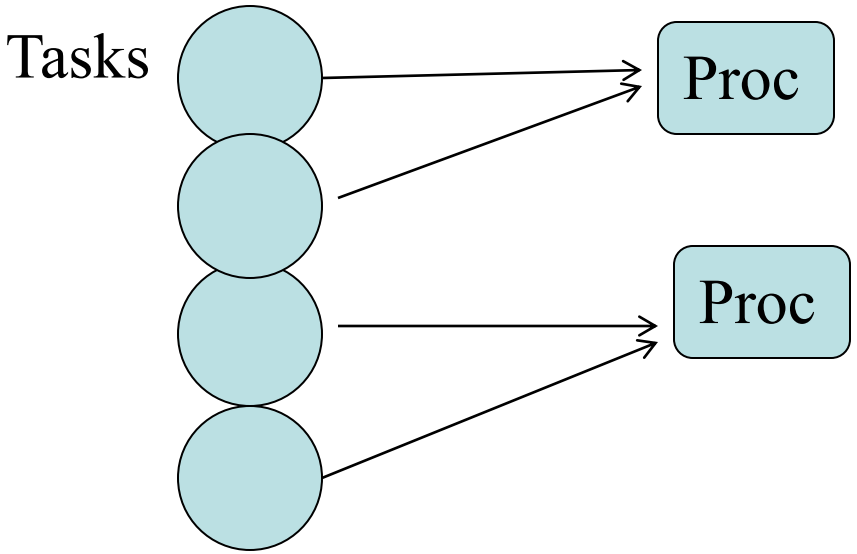


Copyright © 2010, Elsevier Inc. All rights Reserved

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

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