

High-Performance Computing

– Foundations of Parallel Hardware and Parallel Software –

Christian Plessl

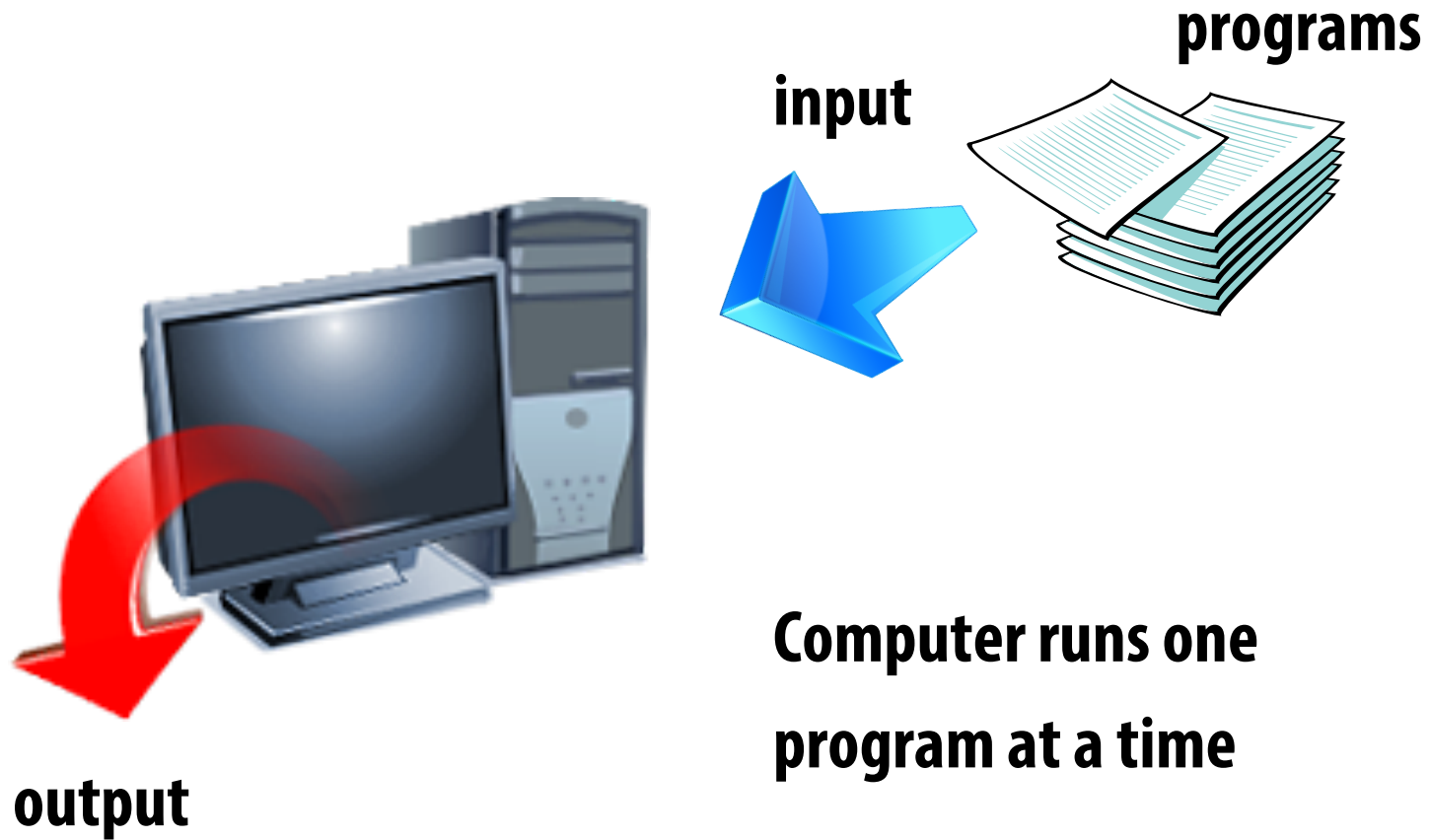
High-Performance IT Systems Group

Paderborn University

Chapter Overview

- **Modifications to the von Neumann model**
 - caching, virtual memory, instruction-level parallelism
- **Parallel hardware**
 - SIMD, MIMD, interconnects, cache-coherence
- **Parallel software**
 - shared and distributed memory programming
- **Input and output**
- **Performance**
 - measurement, models
- **Parallel program design**
 - Foster's methodology

Serial Hardware and Software



The Von Neumann Architecture

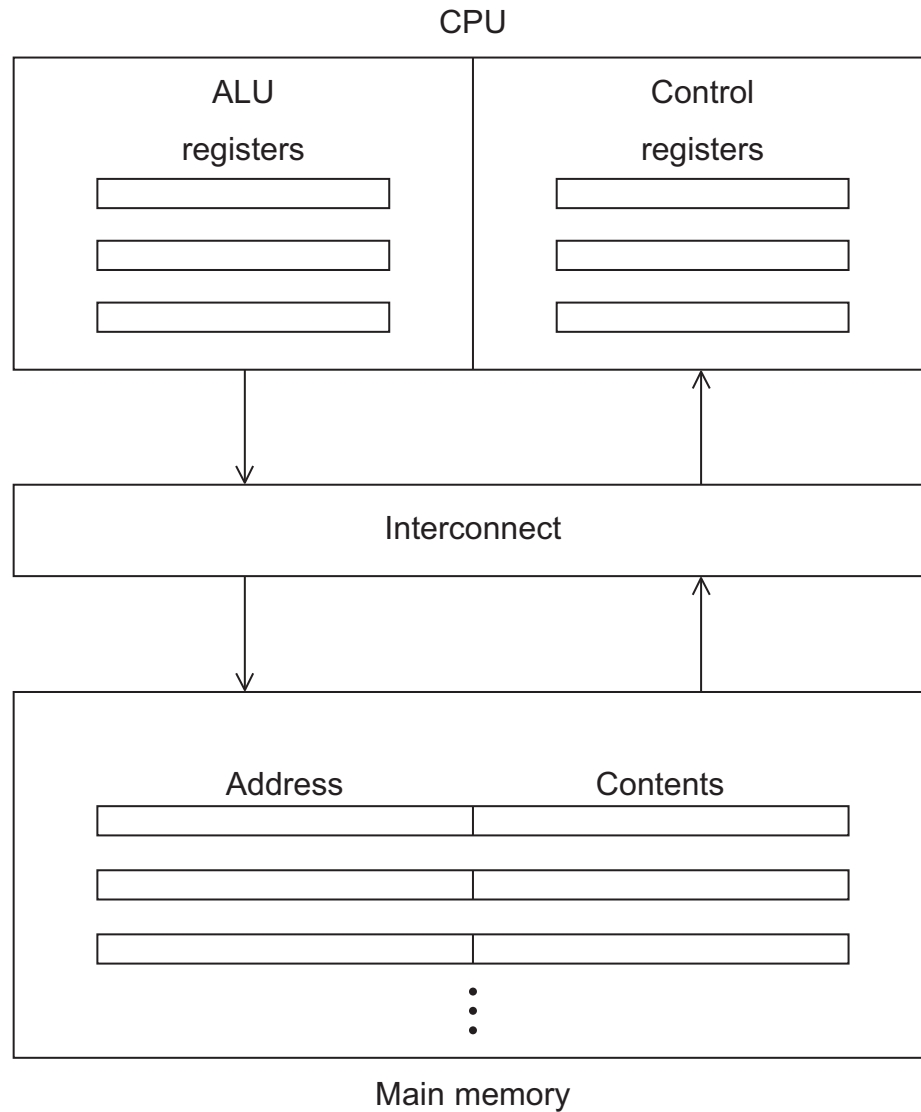


Figure 2.1

Main Memory

- This is a collection of locations, each of which is capable of storing both instructions and data.
- Every location consists of an address, which is used to access the location, and the contents of the location.



Central Processing Unit (CPU)

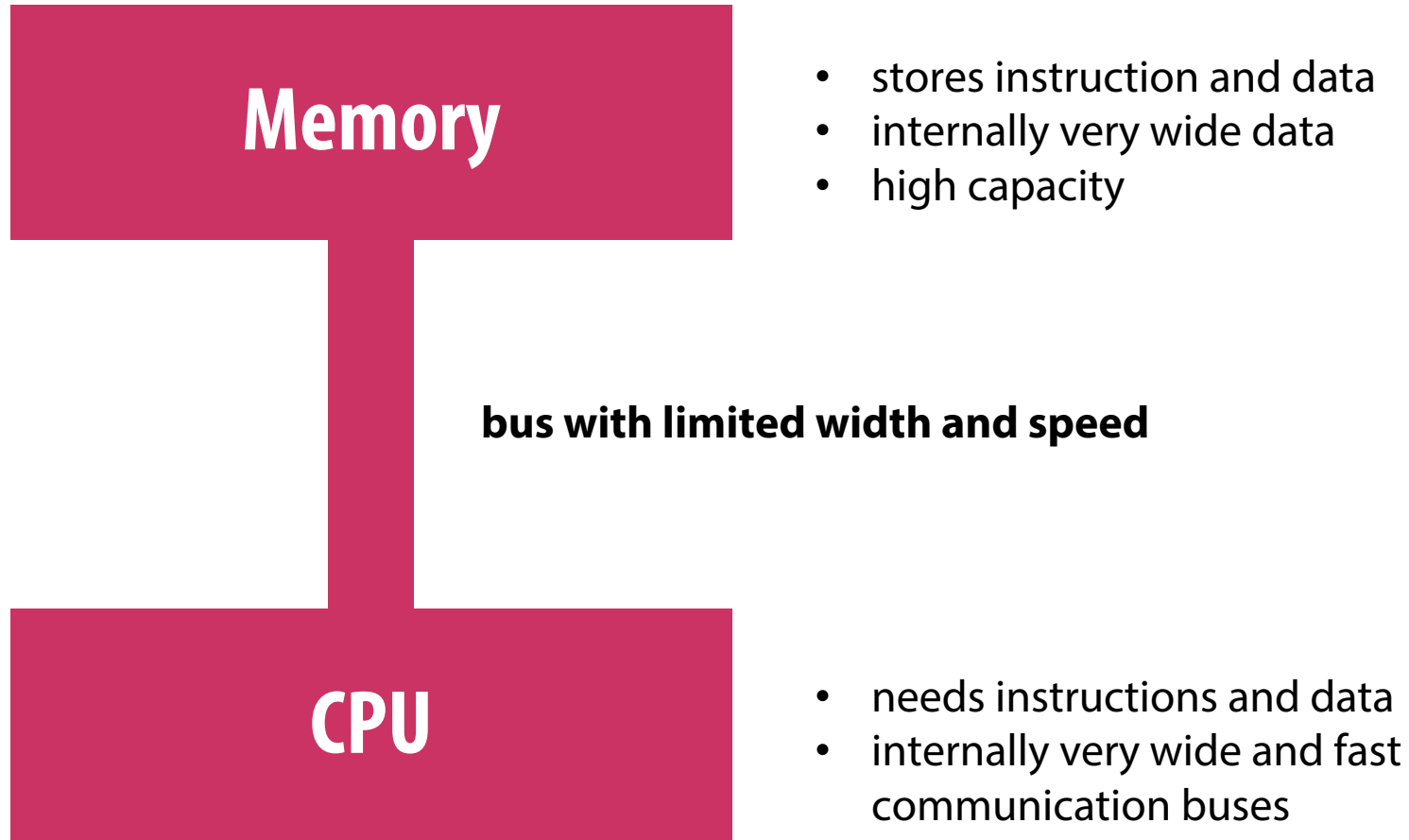
- Divided into two parts
- **Control unit** – responsible for deciding which instruction in a program should be executed (the boss)
- **Arithmetic and logic unit (ALU)** – responsible for executing the actual instructions (the worker)



Key Terms

- **Register** – very fast storage, part of the CPU
- **Program counter** – stores address of the next instruction to be executed
- **Bus** – wires and hardware that connects the CPU and memory

Von Neumann Bottleneck



An Operating System “process”

- **An instance of a computer program that is being executed**
- **Components of a process:**
 - the executable machine language program
 - blocks of memory
 - descriptors of resources the OS has allocated to the process (e.g. files, network connections)
 - security information
 - information about the state of the process

Multitasking

- **Gives the illusion that a single processor system is running multiple programs simultaneously**
- **Two main purposes**
 - **Time slicing**: each process takes turns running, after time quantum is used task blocks until it has a turn again
 - **Throughput improvement**: if a process needs to wait for a busy resource, it is put in a waiting-queue, allowing other processes to progress

Threading

- **Threads are contained within processes**
- **They allow programmers to divide their programs into (more or less) independent tasks**
- **The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run**
- **Main difference of threads and processes**
 - share the process context with all other threads (memory, resources, ...)
 - lightweight: cheap to create and to destroy

One Process and Two Threads

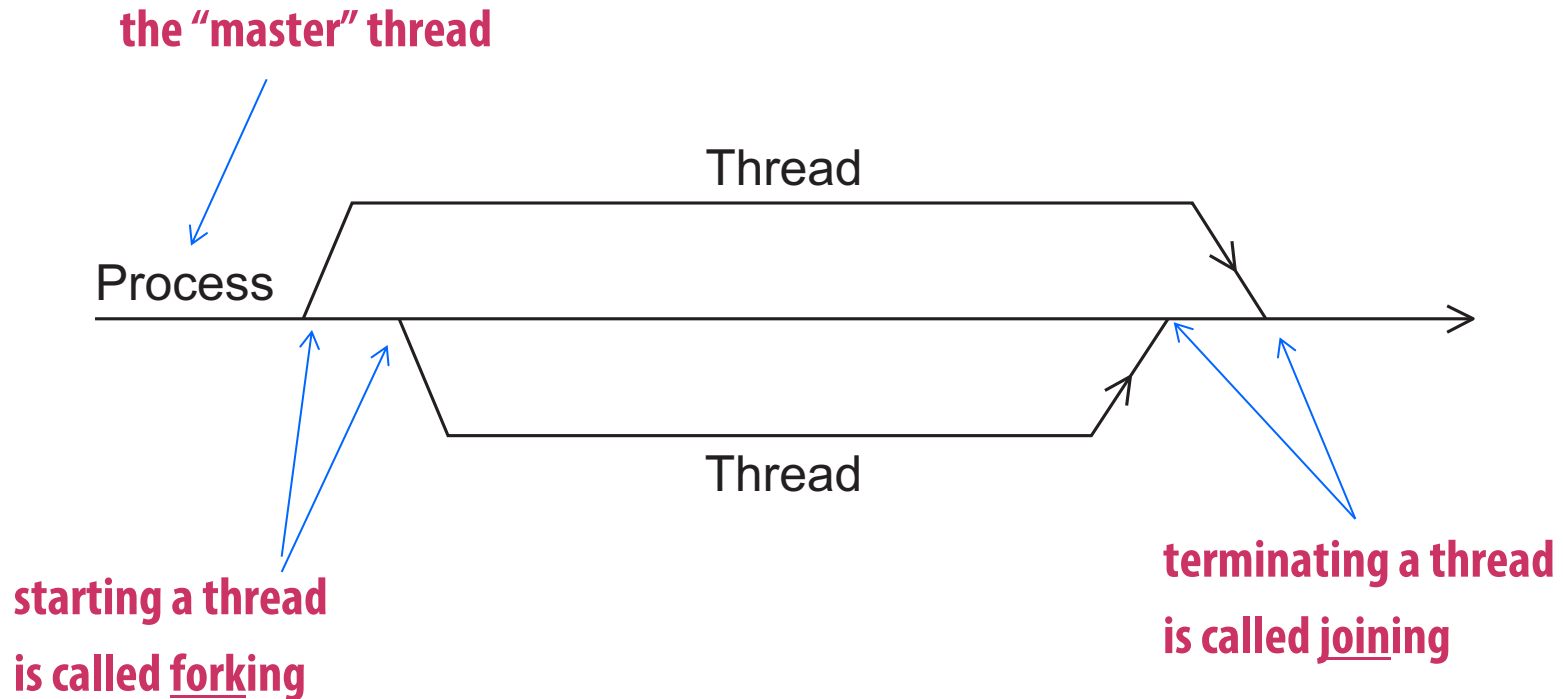


Figure 2.2

Modifications to the Von Neumann Model

- **Techniques to overcome memory bottleneck**
 - reduce the need to access main memory for data (e.g. caching)
 - reduce the need to access main memory for instructions (e.g. caching or SIMD)
 - perform more work per instruction (SIMD, vector processing, ...)

Basics of Caching

- **A collection of memory locations that can be accessed in less time than some other memory locations**
- **A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory**



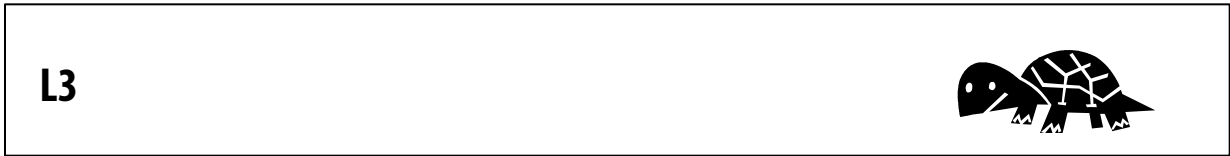
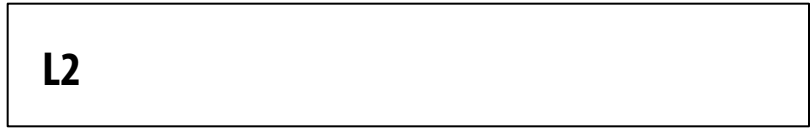
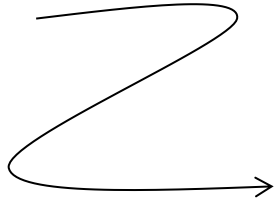
Principle of Locality

- Accessing one location is followed by an access of a nearby location
- **Spatial locality** – accessing a nearby location
- **Temporal locality** – accessing in the near future

```
float z[1000], sum;  
...  
sum = 0.0;  
for (i = 0; i < 1000; i++)  
    sum += z[i];
```

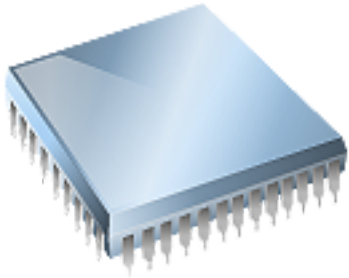
Levels of Cache

smallest & fastest



largest & slowest

Cache Hit



fetch x

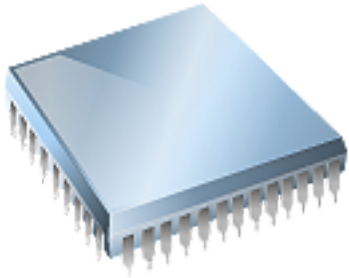


L1	x	sum
----	---	-----

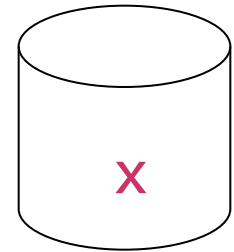
L2	y	z	total
----	---	---	-------

L3	A[]	radius	r1	center
----	------	--------	----	--------

Cache Miss



fetch x



**main
memory**



Issues with Cache

- When a CPU writes data to cache, the value in cache may be **inconsistent** with the value in main memory
- **Write-through caches** handle this by updating the data in main memory at the time it is written to cache
- **Write-back caches** mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory

Cache Associativity

- **Full associative**

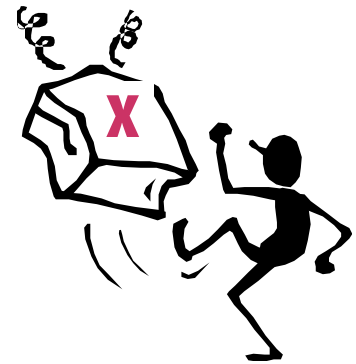
- a new line can be placed at any location in the cache

- **Direct mapped**

- each cache line has a unique location in the cache to which it will be assigned

- **n-way set associative**

- each cache line can be placed in one of n different locations in the cache
- if a cache line needs to be replaced or evicted, the cache needs to decide which one



Example

Memory Index	Possible placement in cache		
	full assoc	direct mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Caches and Programs

```
double A[MAX][MAX], x[MAX], y[MAX];
...
/* Initialize A and x, assign y = 0 */
...
/* First pair of loops */

for (i=0; i<MAX; i++)
    for (j=0; j<MAX; j++)
        y[i] += A[i][j]*x[j];

/* Assign y=0 */

/* First pair of loops */

for (j=0; j<MAX; j++)
    for (i=0; i<MAX; i++)
        y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

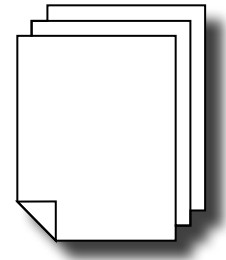
Example for MAX=4

Virtual Memory (1)

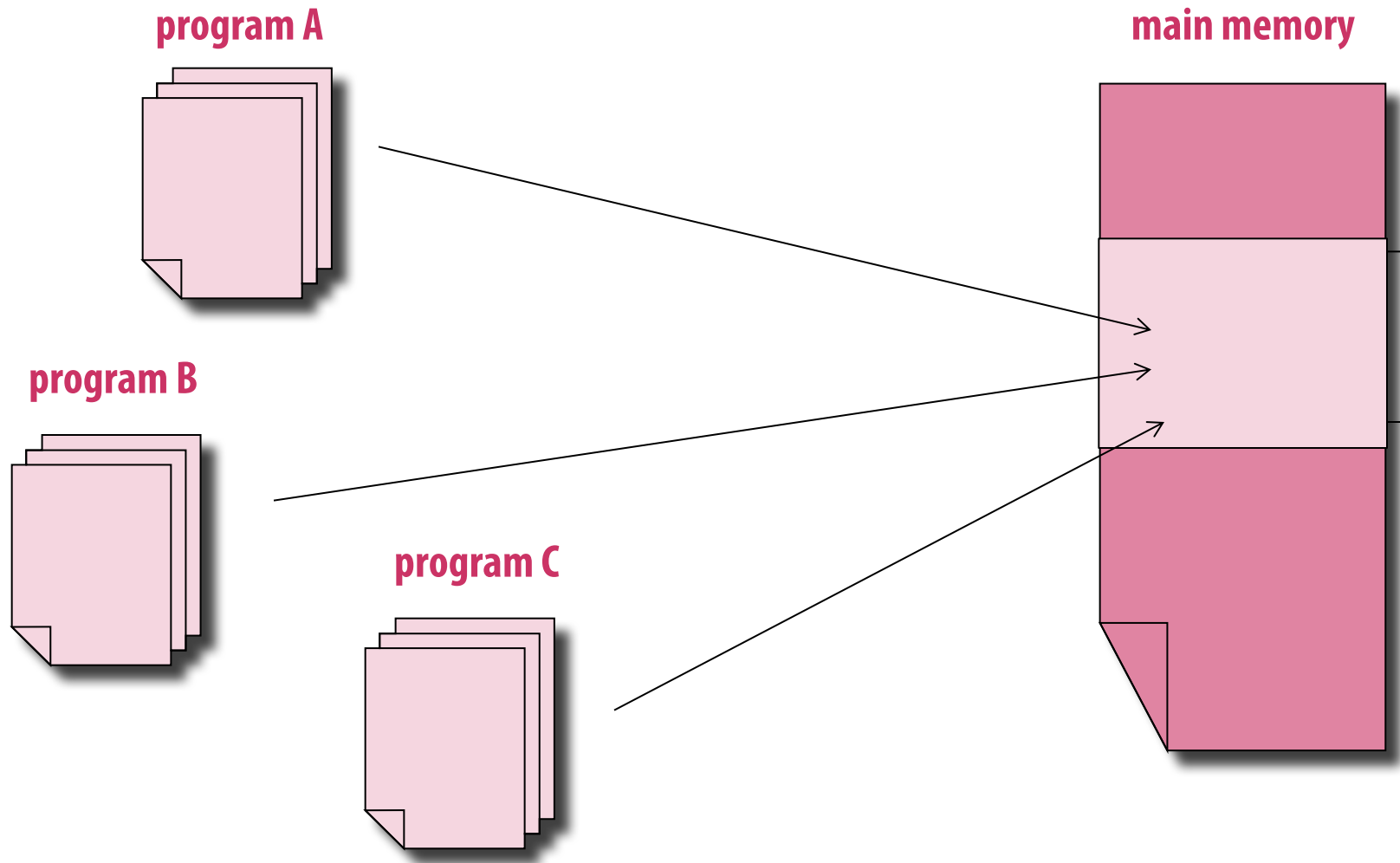
- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory
- Virtual memory functions as a **cache for secondary storage**
- It exploits the principle of spatial and temporal locality (like caches)
- It only keeps the active parts of running programs in main memory

Virtual Memory (2)

- **Swap space** - those parts that are idle are kept in a block of secondary storage
- **Pages** – blocks of data and instructions
 - usually these are relatively large
 - most systems have a fixed page size that currently ranges from 4 to 16 kilobytes



Virtual Memory (2)



Virtual Addresses and Page Table

- When a program is **compiled** it uses **virtual addresses** that are assigned to **virtual page numbers**

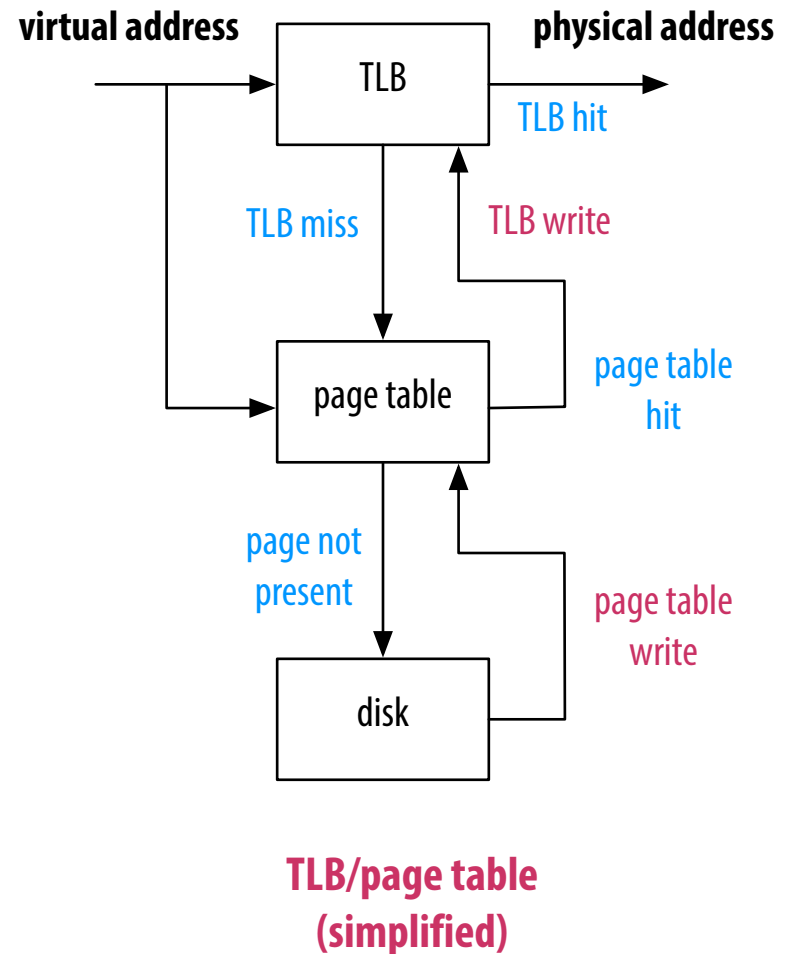
Virtual Address									
Virtual Page Number					Byte Offset				
31	30	...	13	12	11	10	...	1	0
1	0	...	1	1	0	0	...	1	1

Table 2.2: Virtual Address Divided into Virtual Page Number and Byte Offset (4k Page Size)

- When the program is run, a table is created that maps the virtual page numbers to physical addresses
- A **page table** is used to translate the virtual address into a physical address

Translation-Lookaside Buffer (TLB)

- **Using a page table adds a layer of indirection for each load/store operation**
 - would significantly increase the program's overall run-time
- **Translation-lookaside buffer**
 - special **address translation cache** in the processor
 - caches a small number of entries (typically 16–512) from the page table in very fast memory
- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk



Instruction Level Parallelism (ILP)

- **Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions**
- **Two approaches**
 - **Pipelining** – overlapped processing of instructions by functional units arranged in stages
 - **Multiple issue** – simultaneously initiation and execution of multiple instructions

Pipelining (1)

- Divide processing into a sequence of steps
- Pass intermediate data between pipeline stages
- Example:

Add the floating point numbers 9.87×10^4 and 6.54×10^3

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Pipelining (2)

- Assume each operation takes one nanosecond (10^{-9} seconds)
- This for loop takes about 7000 nanoseconds

```
float x[1000], y[1000], z[1000];  
...  
for ( i = 0; i < 1000; i++)  
    z[i] = x[i] + y[i];
```

- Divide the floating point adder into 7 separate pieces of hardware or functional units
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next

Pipelining (3)

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Table 2.3: Pipelined Addition.

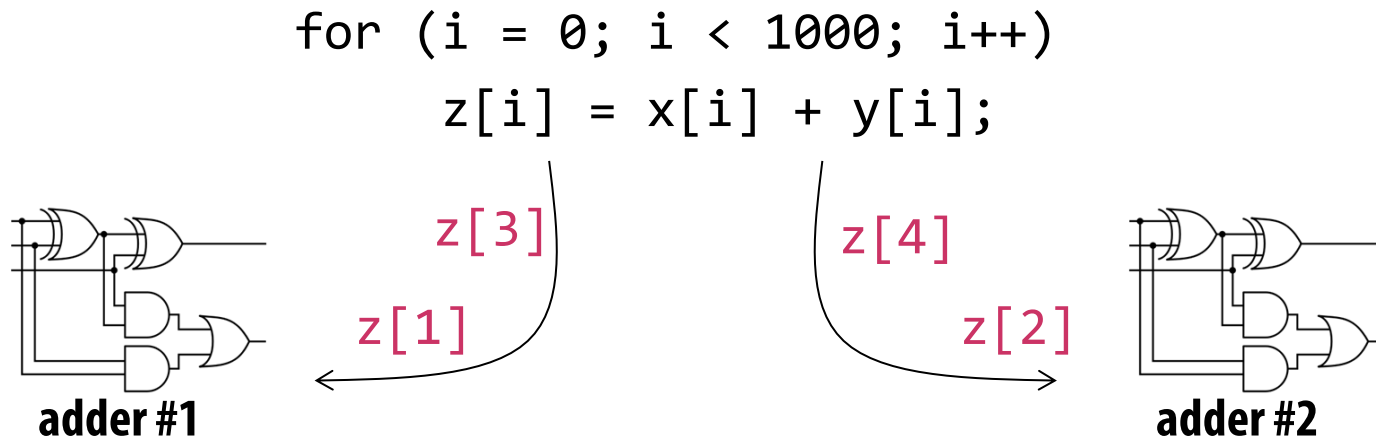
Numbers in the table are subscripts of operands/results

Pipelining (4)

- **Latency of operations is unchanged**
 - one floating point addition still takes 7 nanoseconds
- **Increase throughput**
 - 1000 floating point additions now takes 1006 nanoseconds
 - pipeline with N pipeline stages ideally achieves a speedup of N
 - challenges
 - balancing of pipeline stages
 - data dependencies

Multiple Issue

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program

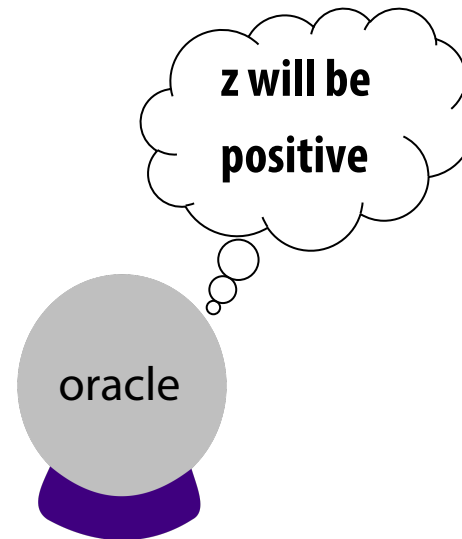


- static multiple issue (VLIW)** – functional units are scheduled at compile time
- dynamic multiple issue (superscalar)** – functional units are scheduled at run-time

Speculation

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously
- Approach: compiler or processor make guesses about the outcome of control-flow decisions and execute instructions on the basis of the guess

```
z = x + y ;  
if ( z > 0 )  
    w = x ;  
else  
    w = y ;
```

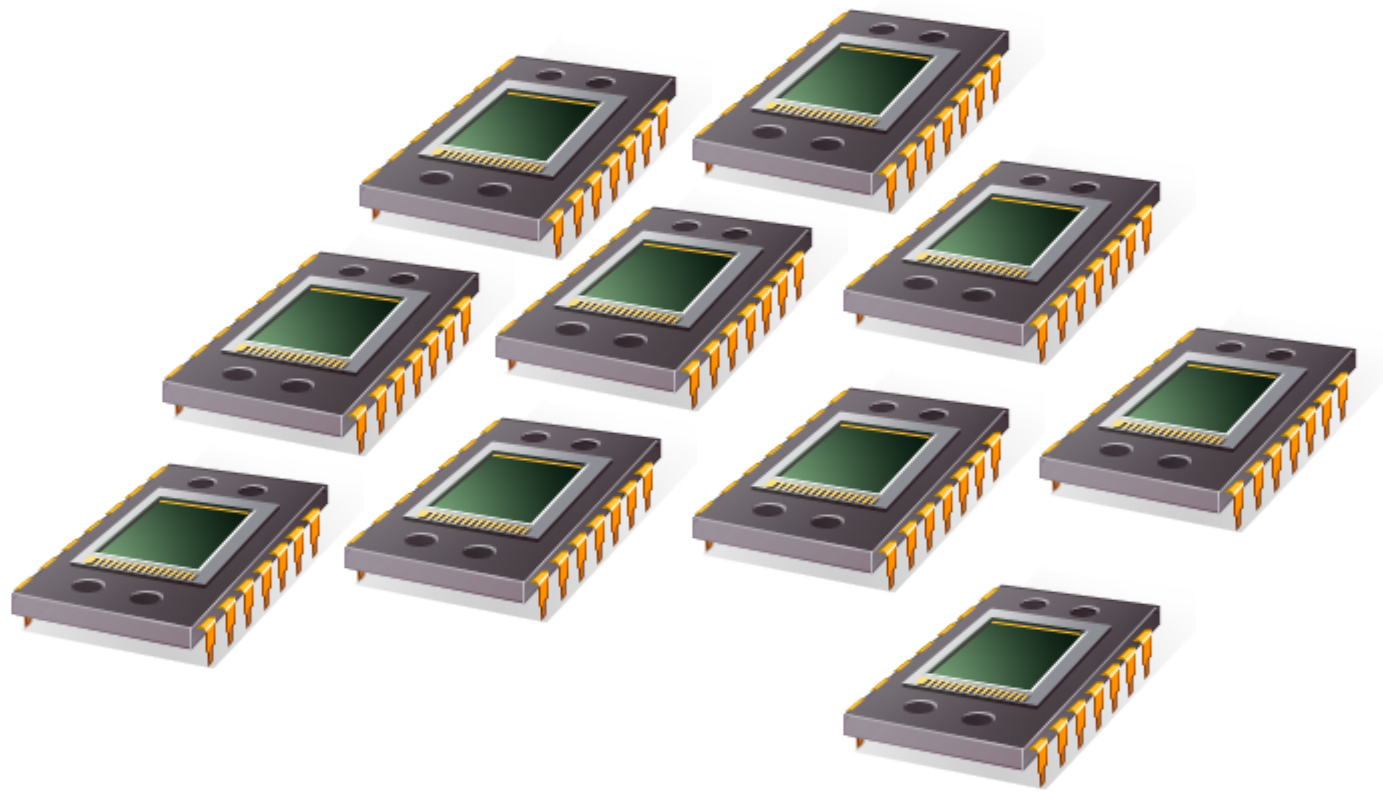


If the system speculates incorrectly, it must go back and recalculate $w = y$

Hardware Multithreading

- **Hardware multithreading allows CPU to continue doing useful work when the task being currently executed has stalled**
 - Ex., the current task has to wait for data to be loaded from memory
- **Fine-grained – the processor switches between threads after each instruction, skipping threads that are stalled**
 - Pro: potential to avoid wasted machine time due to stalls
 - Con: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction
- **Coarse-grained – only switches threads that are stalled waiting for a time-consuming operation to complete.**
 - Pros: switching threads doesn't need to be nearly instantaneous
 - Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays
- **Simultaneous multithreading (SMT) - a variation on fine-grained multithreading**
 - issues instructions from different threads in every cycle
 - hence, instructions from different threads can be concurrently executed in each pipeline stage

Explicitly Exploitation of Parallelism



Flynn's Taxonomy

- Classification of Parallel Computer Architectures

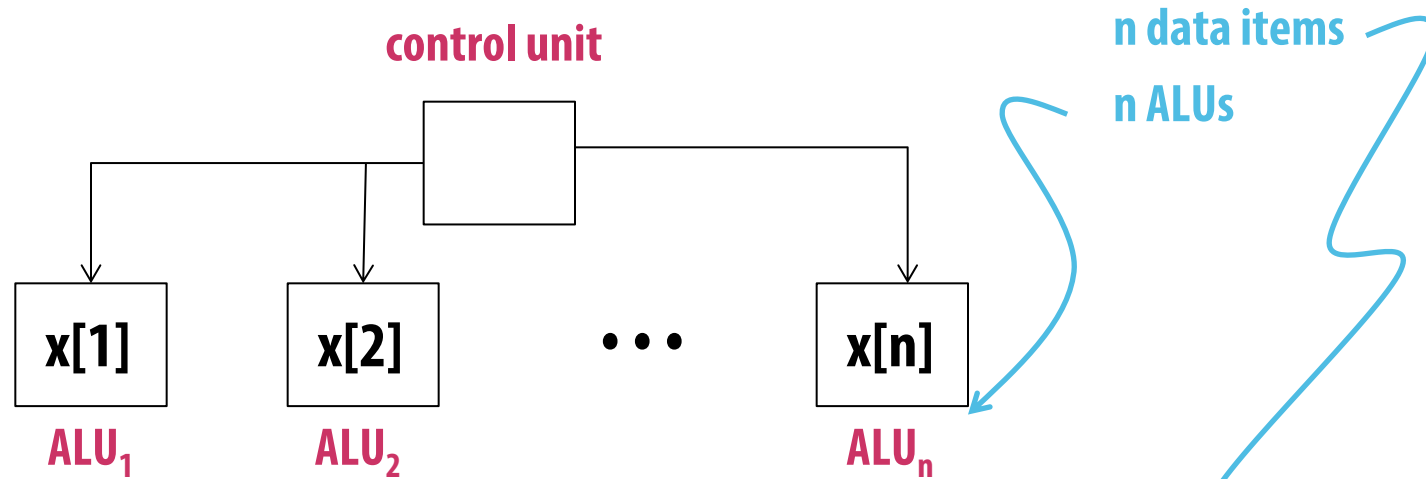
<p>SISD</p> <p>Single instruction stream Single data stream classic Von Neumann</p>	<p>SIMD</p> <p>Single instruction stream Multiple data stream</p>
<p>MISD</p> <p>Multiple instruction stream Single data stream</p>	<p>MIMD</p> <p>Multiple instruction stream Multiple data stream</p>

Michael J. Flynn, Kevin W. Rudd. **Parallel Architectures** CRC Press, 1996.

SIMD

- **Parallelism achieved by dividing data among the processing elements**
- **Applies the same instruction to multiple data items**
- **Denoted as **data parallelism****

SIMD Example



```
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

SIMD

- What if we don't have as many ALUs as data items?
- Divide the work and process iteratively
- Ex. $m = 4$ ALUs 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]	

SIMD Pros and Cons

■ Advantages

- simple extension to scalar CPU core
- effective method for data parallel computations

■ Disadvantages

- restricted to highly regular code without control flow, because ALUs are required to execute the same instruction, or remain idle
- requires code modification to be exploited
- SIMD units are only useful for data parallel code, not for other forms of parallelism
- challenge for memory subsystem supply SIMD units with sufficient data

Vector Processors

- **Operate on arrays or vectors of data while conventional CPUs operate on individual data elements or scalars**
- **Everything is vectorized**
 - registers: can store a vector of operands and allows concurrent access to contents
 - functional units: same operation is applied to each element in the vector (or pairs of elements)
 - instructions: operate on vectors rather than scalars
- **Vector length can be wider than functional units**
 - difference to SIMD instructions
- **Parallel memory subsystem**
 - multiple memory banks for concurrent access
 - interleaving schemes to distribute vectors across banks, reduce or eliminate delay in loading/storing successive elements
 - support for efficient strided access and scatter/gather

Vector processors – Pros and Cons

■ Advantages

- very effective for scientific codes using dense linear algebra
- vectorizing compilers are good at automatically identifying vectorizable code
- effective use of memory bandwidth and caches (uses every item in a cache line)

■ Disadvantages

- handling of control-flow and irregular data structures is inefficient
- limited scalability
 - CPU implementation imposes physical limits on width of vector processing units
 - finite length of vectors in applications
- data or control-flow dependencies may prohibit auto-vectorization
 - manual refactoring of code needed

Graphics Processing Units (GPU)

- **Developed for real-time processing of geometric primitives (points, lines, triangles) that represent the surface of objects**
- **Graphics processing pipeline**
 - converts internal representation into an array of pixels that can be sent to a computer screen
 - initially a strict pipeline of fixed functions
 - since early 2000's stages of this pipeline (shaders) have become more and more programmable
- **Programmable Shaders**
 - typically just a few lines of C code, working on geometric primitives
 - execution on primitives is implicitly parallel
 - can implement linear algebra functions
 - gave rise to application in high-performance computing

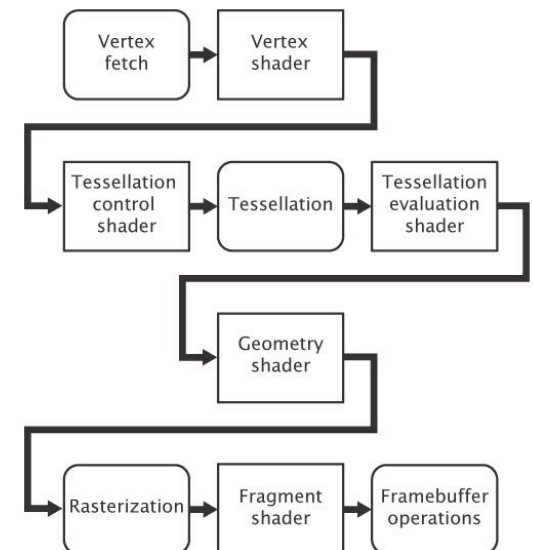
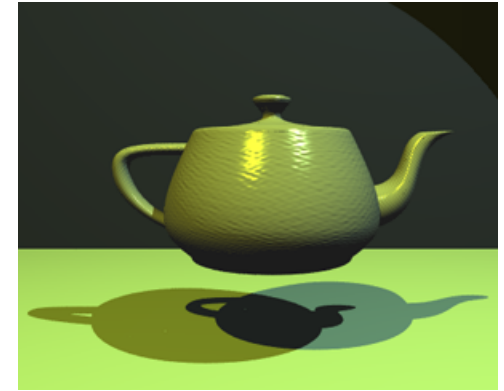


Figure 1.1: Simplified graphics pipeline

GPUs (2)

■ General-Purpose GPUs (GPGPUs)

- further generalized processing pipeline
- directly programmable with computing-focused programming languages (e.g. CUDA, OpenCL)

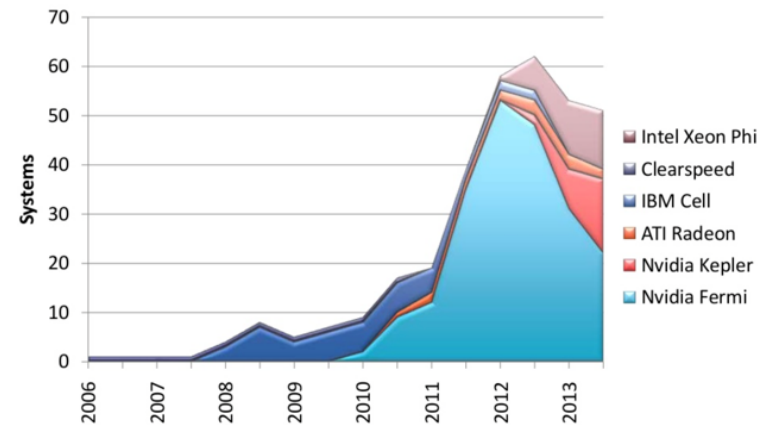
■ Current GPUs have a hybrid architecture

- combination of SIMD and hardware multi-threading (SIMT)
- increasingly MIMD-like architecture

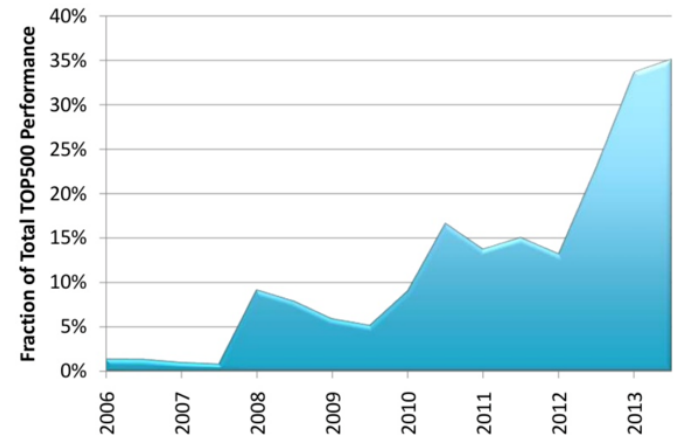
■ Most widely used accelerator in HPC

- supported by increasing number of scientific codes

Accelerators



Performance Share of Accelerators



use of accelerators in 500 most powerful supercomputers

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**
- **Multi-Core CPUs are currently the most widespread MIMD architectures**
 - although each core typically follows a SIMD model

Shared Memory System (1)

- **A collection of autonomous processors is connected to a memory system via an interconnection network.**
 - each processor can access each memory location (same address space)
 - processors usually communicate implicitly by accessing shared data structures
- **Most widely available shared memory systems use one or more multicore processors**

Shared Memory System (2)

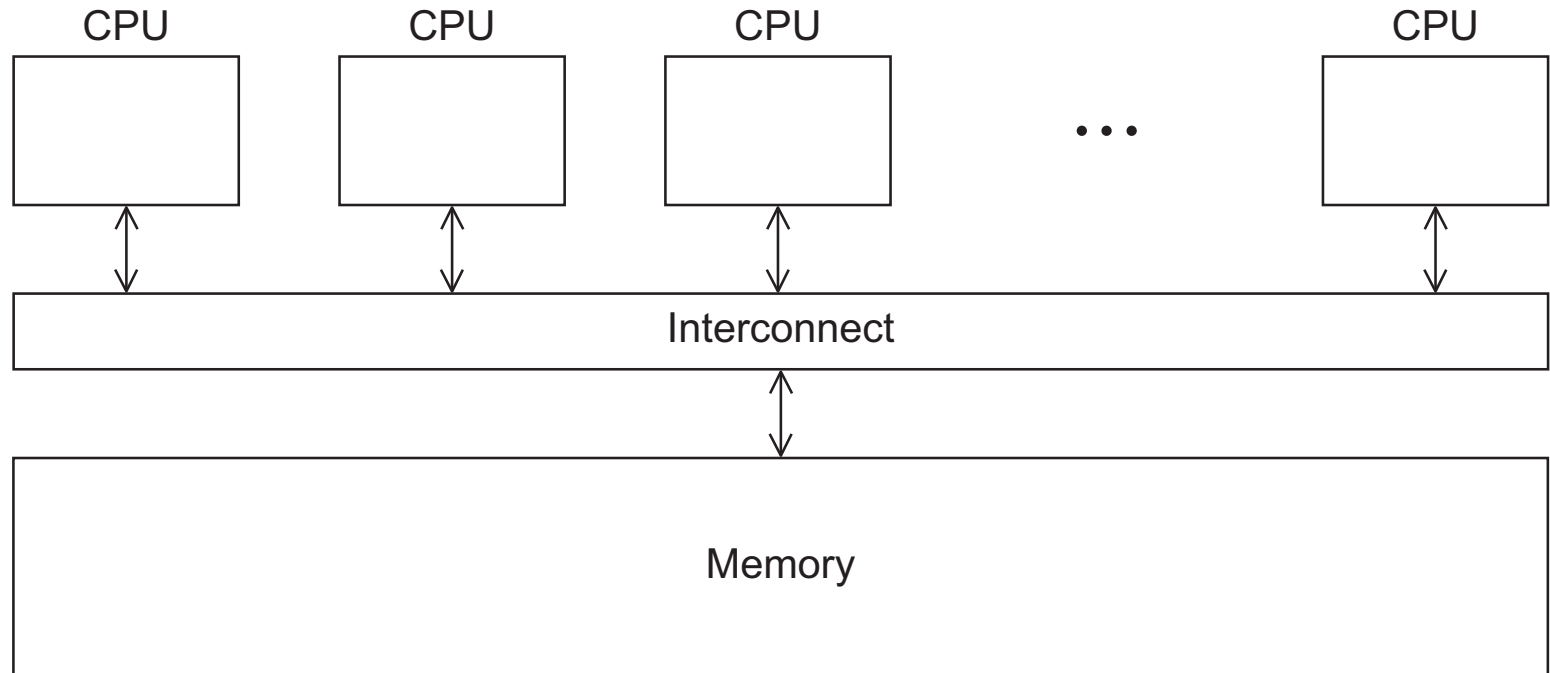


Figure 2.3

Unified Memory Access (UMA) Multicore System

- Access time and latency to access all the memory locations is the same for all cores

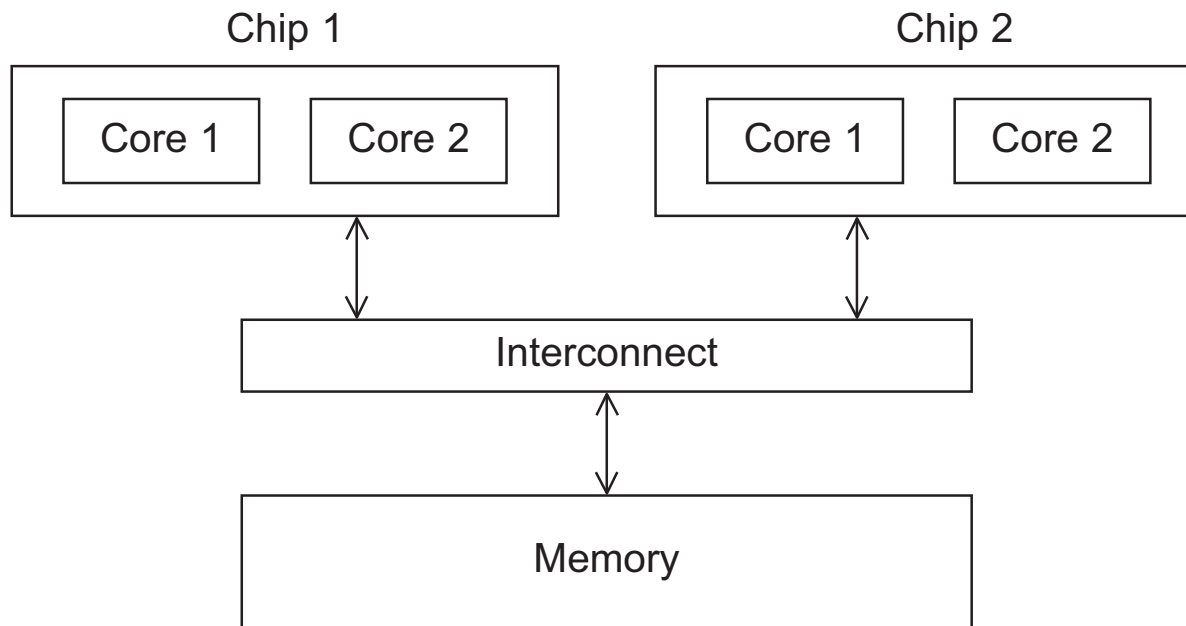


Figure 2.5

Non-Unified Memory Access (NUMA) Multicore System

- Still, all cores share **same address space**
- But cores can **access locally connected memory faster** than memory connected through another processor
- Requires **careful data placement** for optimal memory subsystem performance

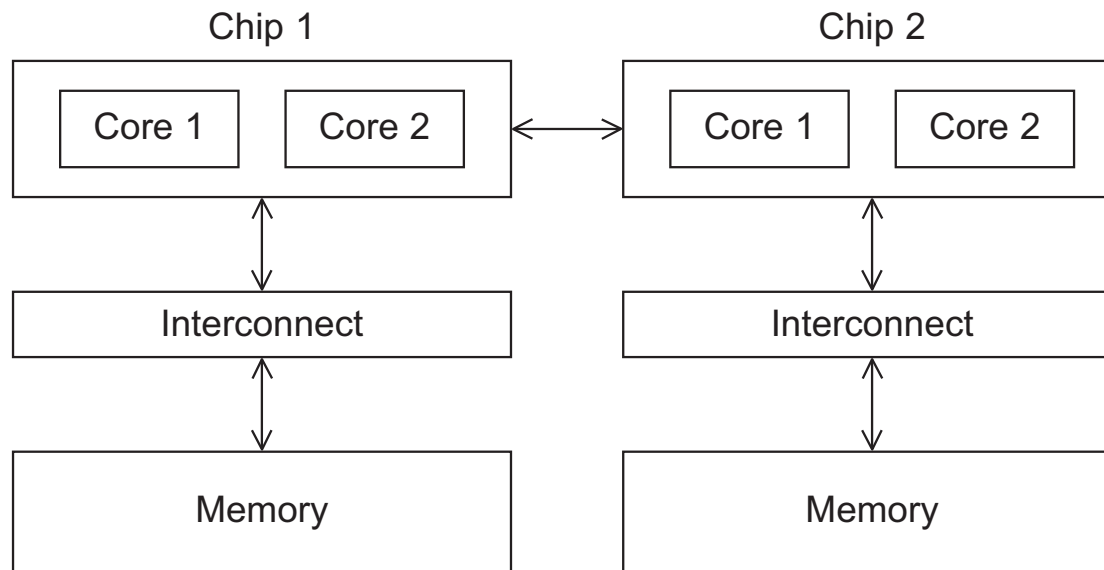


Figure 2.6

Distributed Memory System

■ HPC Computer Clusters

- collection of **commodity servers**
- connected by a **commodity network** (InfiniBand, Ethernet)
- most popular and cost-effective HPC systems today

■ Cluster nodes are independent

- resource management system and communication library allow system to be used as one unit

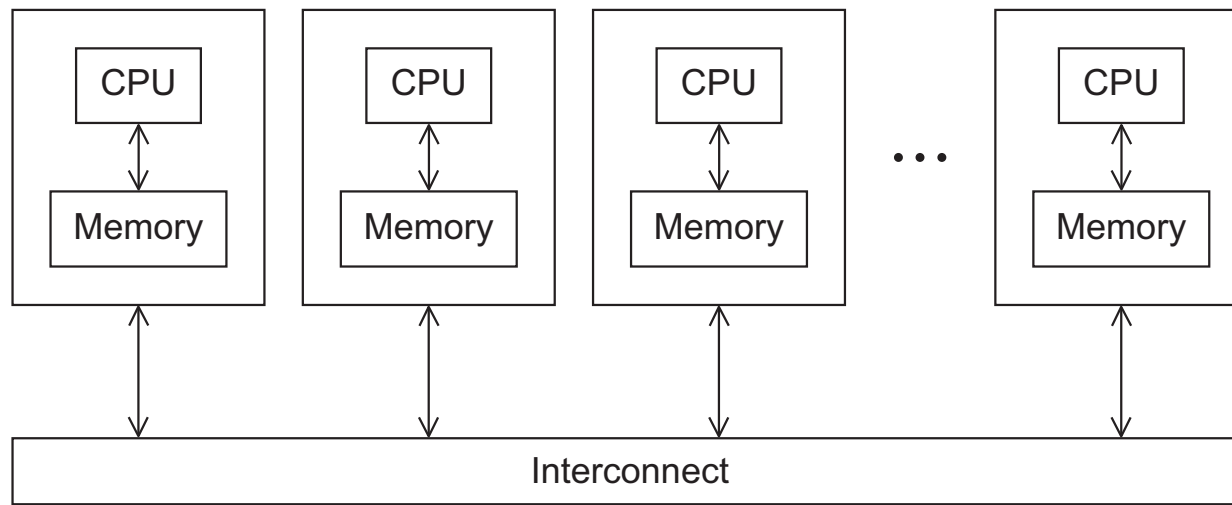


Figure 2.4

Interconnection networks

- **Choice of interconnection network determines performance of both distributed and shared memory systems**
 - Tradeoff between hardware effort / cost and performance
- **Two categories:**
 - Shared memory interconnects
 - Distributed memory interconnects

Shared Memory Interconnects

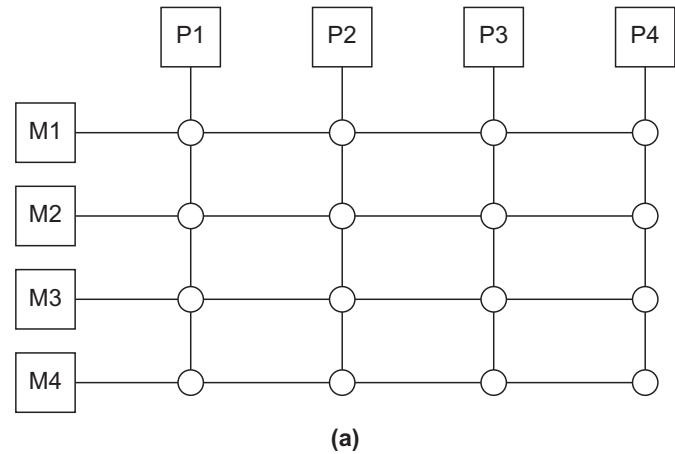
■ Bus interconnect

- shared parallel communication wires connect communicating units
- arbiter controls access to bus
- with increasing number of devices, contention for use of the bus increases, and performance decreases

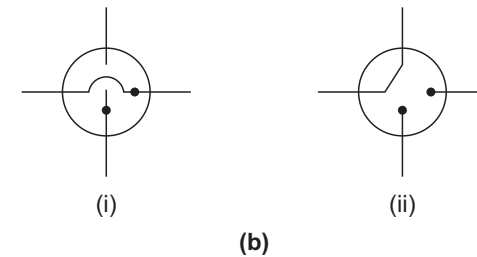
■ Switched interconnect

- uses switches to control the routing of data among the connected devices
- crossbar
 - allows simultaneous communication among different devices
 - faster than buses
 - cost of the switches and links is relatively high

(a) crossbar switch connecting 4 processors (P_i) and 4 memory modules (M_j)



(b) configuration of internal switches in a crossbar



(c) simultaneous memory accesses by the processors

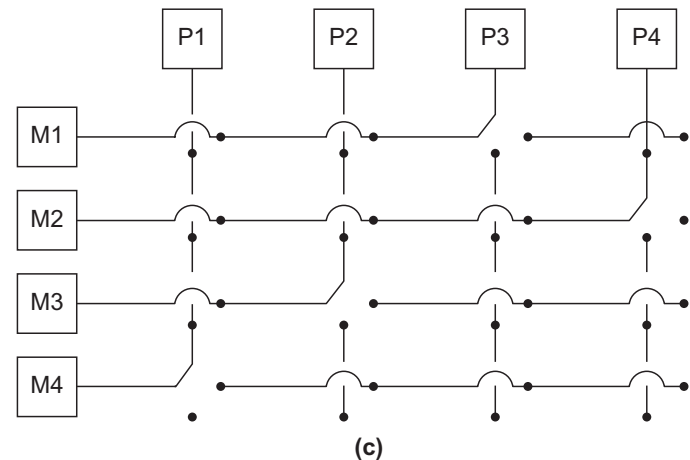


Figure 2.7

Distributed Memory Interconnects

- **Direct interconnect**

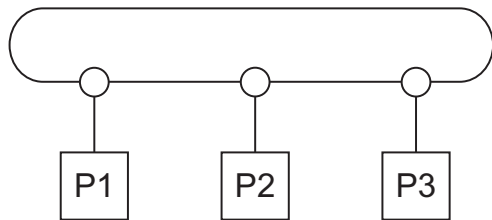
- each switch is directly connected to a processor memory pair, and the switches are connected to each other

- **Indirect interconnect**

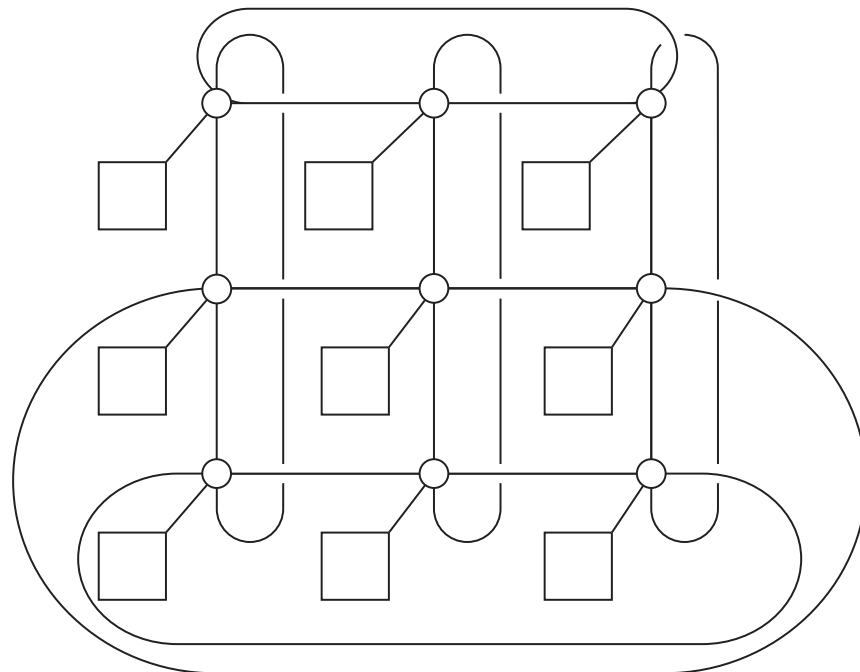
- switches may not be directly connected to a processor

Direct Interconnect

Figure 2.8



(a)
ring



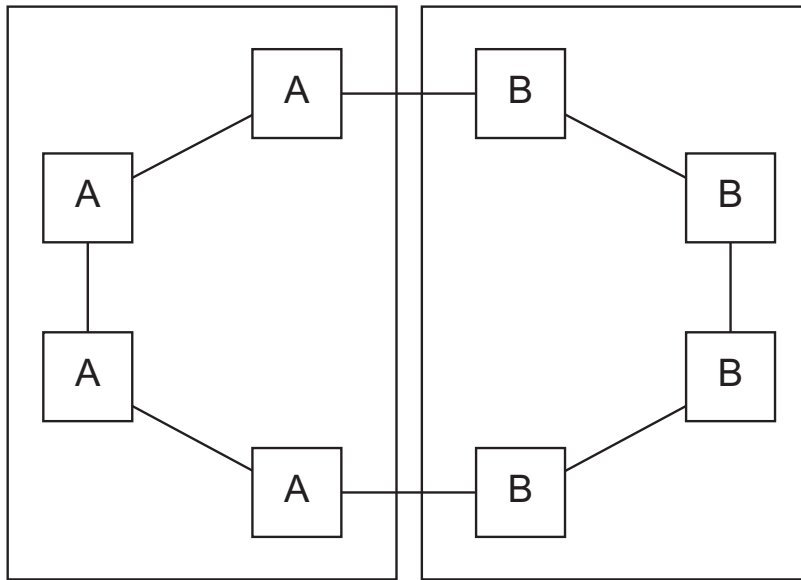
(b)
toroidal mesh

Bisection Width

- A measure of “number of simultaneous communications” or “connectivity”
- **Computing the bisection width**
 - How many simultaneous communications can take place “across the divide” between the halves?
 - Or, how many connections need to be removed to split the topology into two halves?

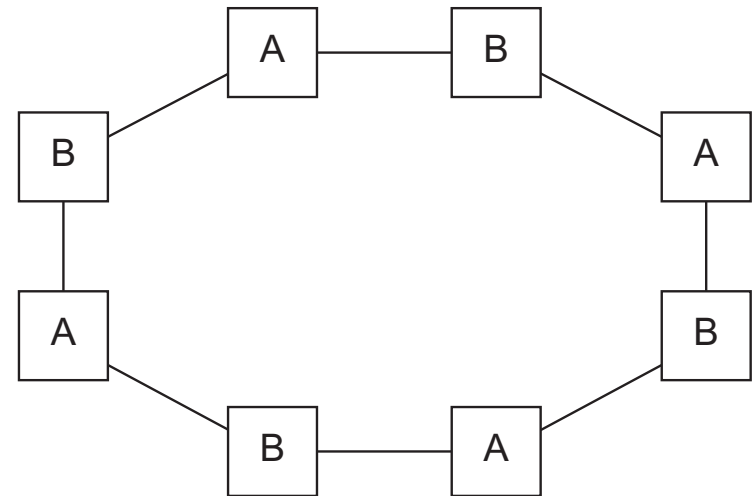


Two Bisections of a Ring



(a)

removing two connections
creates a bi-partition



(b)

removing eight connections
creates bi-partition

Figure 2.9

A Bisection of a Toroidal Mesh

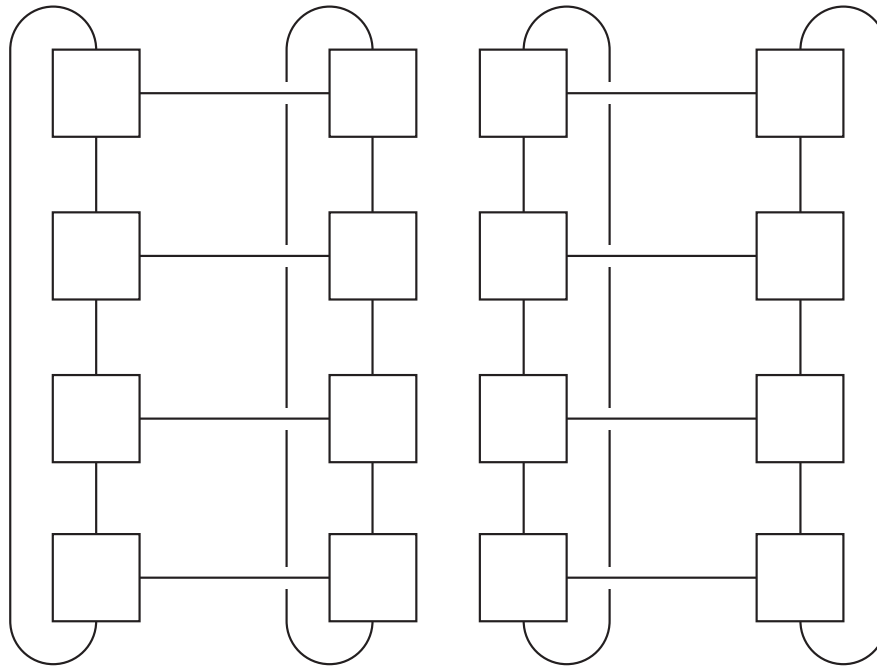


Figure 2.10

Definitions

■ **Bandwidth**

- the rate at which a link can transmit data
- usually given in megabits or megabytes per second

■ **Bisection bandwidth**

- a measure of network quality
- instead of counting the number of links joining the halves, it sums the bandwidth of the links

Fully Connected Network

- Each switch is directly connected to every other switch

bisection width = $p^2/4$

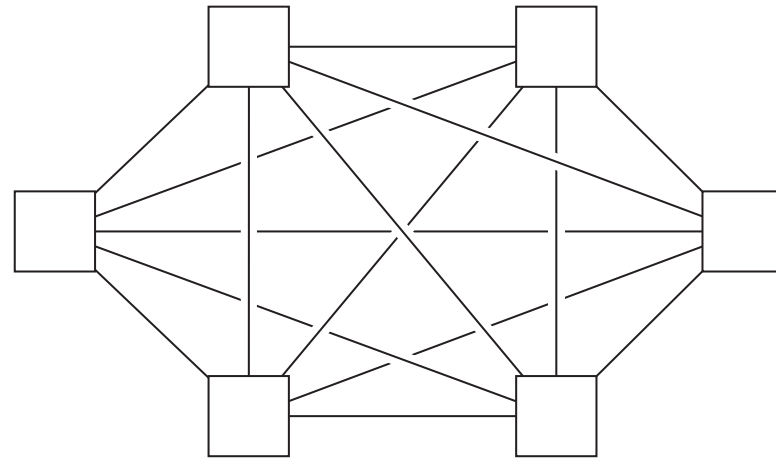


Figure 2.11

impractical: $p^2/2 - p/2$ links required; each switch needs p ports

Hypercube

- **Highly connected direct interconnect**
- **Built inductively**
 - 1D hypercube is a fully-connected system with two processors
 - 2D hypercube is built from two 1D hypercubes by joining “corresponding” switches
 - 3D hypercube is built from two 2D hypercubes

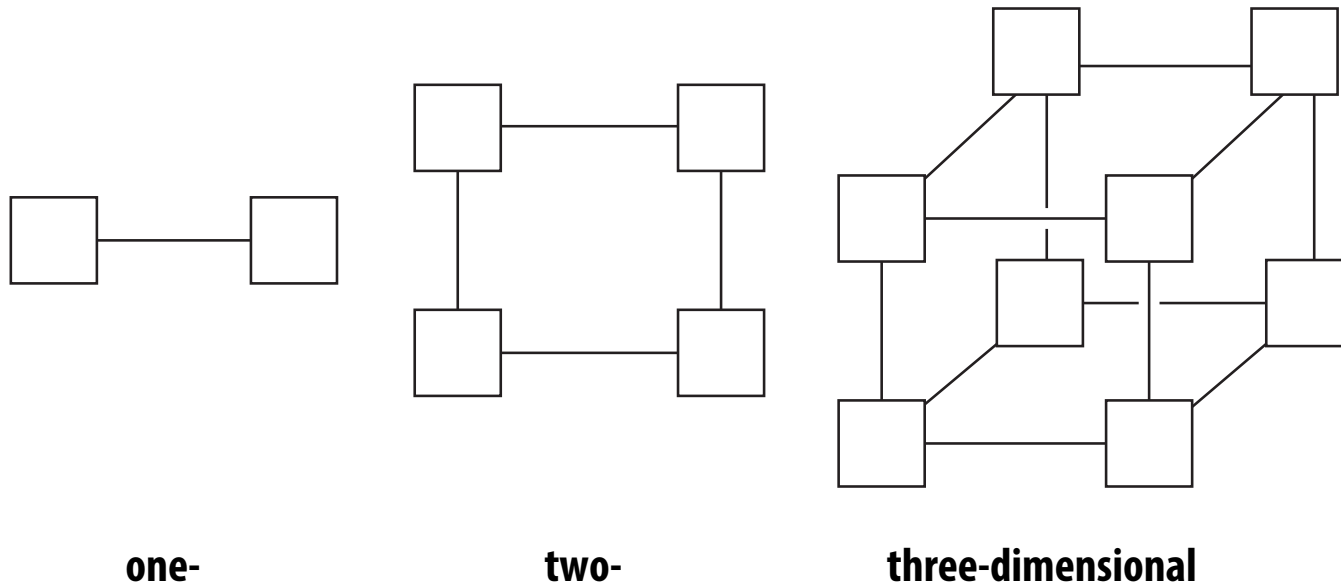


Figure 2.12

Indirect Interconnects

- **Simple examples of indirect networks**
 - Crossbar
 - Omega network
- **Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network**

Generic Indirect Network

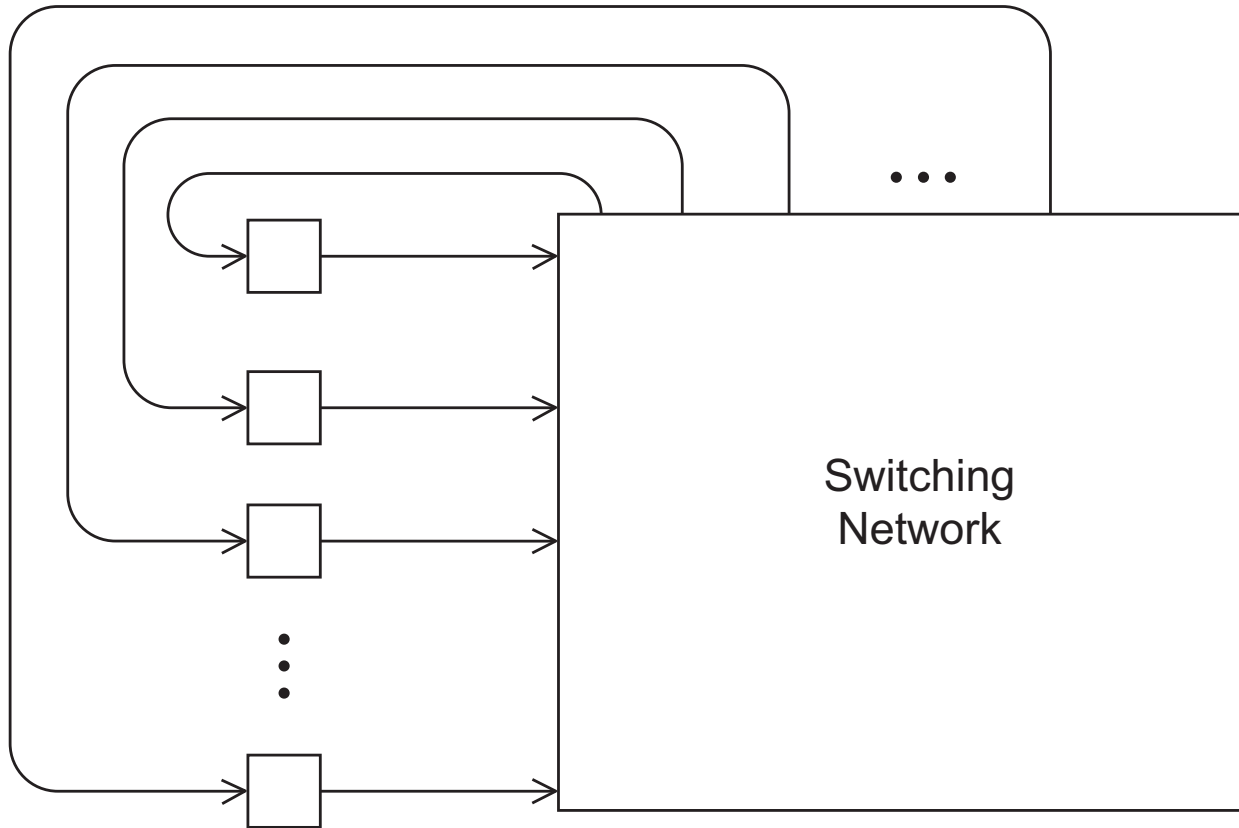


Figure 2.13

Crossbar Interconnect for Distributed Memory

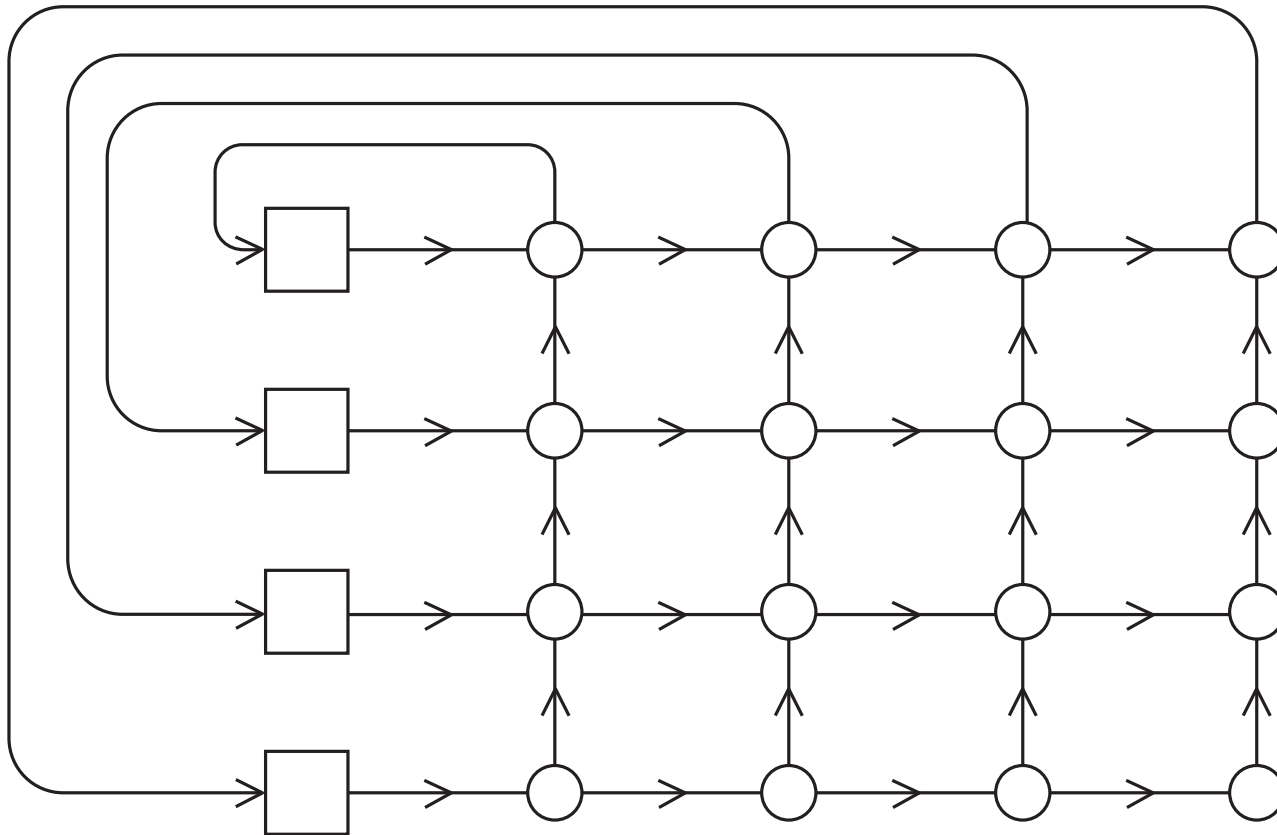


Figure 2.14

Omega Network

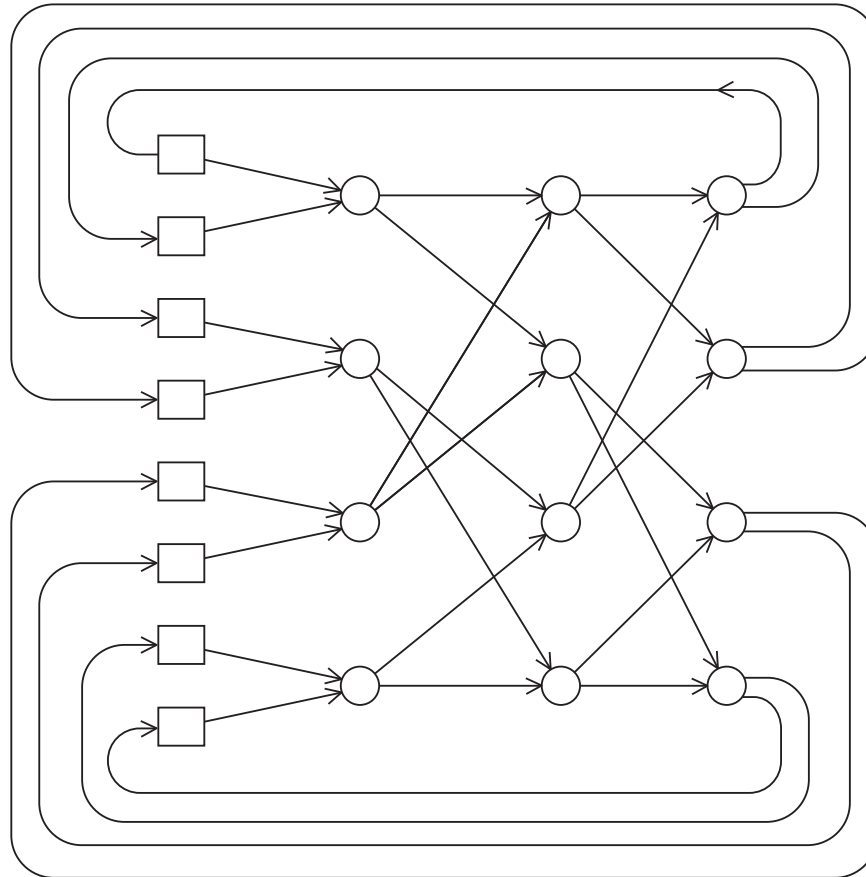


Figure 2.15

A Switch in an Omega Network

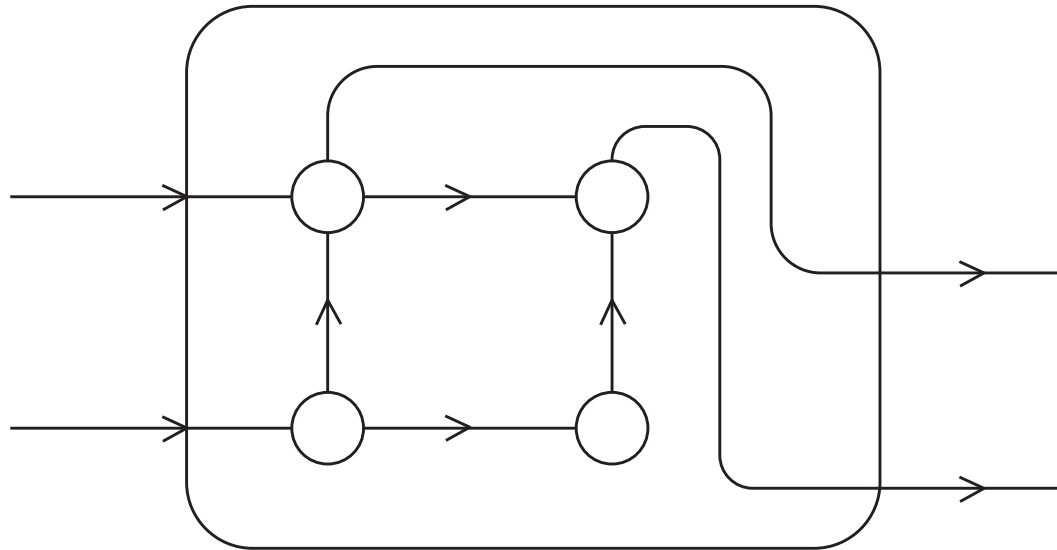


Figure 2.16

More definitions

- **Any time data is transmitted, we're interested in how long it will take for the data to reach its destination**
- **Latency**
 - time that elapses between the source beginning to transmit the data and the destination starting to receive the first byte
- **Bandwidth**
 - the rate at which the destination receives data after it has started to receive the first byte

$$\text{message transmission time} = l + n / b$$

l: latency (seconds)

n: length of message (bytes)

b: bandwidth (bytes per second)

Cache Coherence (1)

- **Programmers have no control over caches and when they get updated**

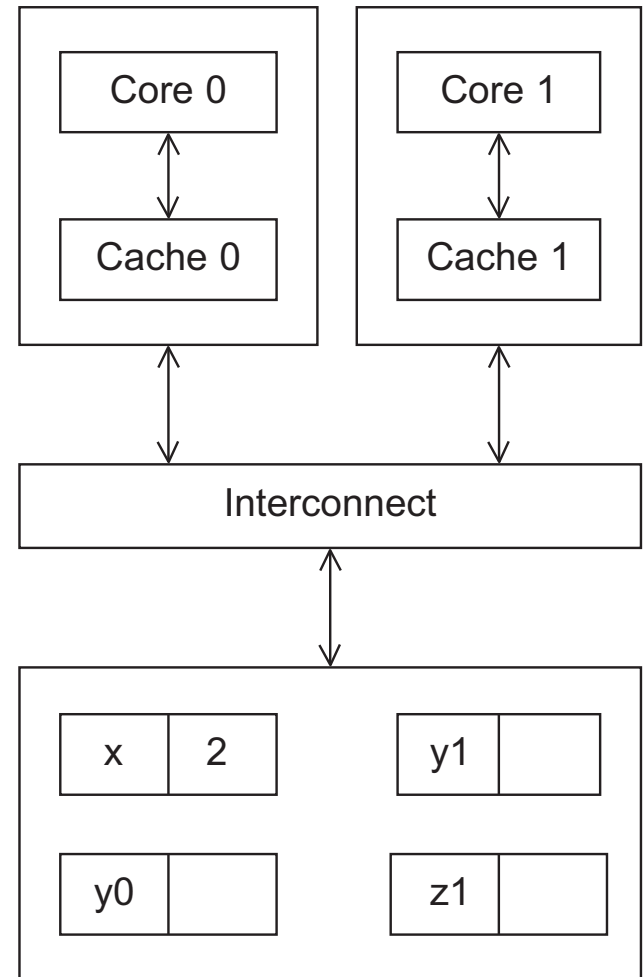


Figure 2.17

A shared memory system with two cores and two caches

Cache Coherence (2)

- Programmers have no control over caches and when they get updated

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

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 = ???

A shared memory system with two cores and two caches

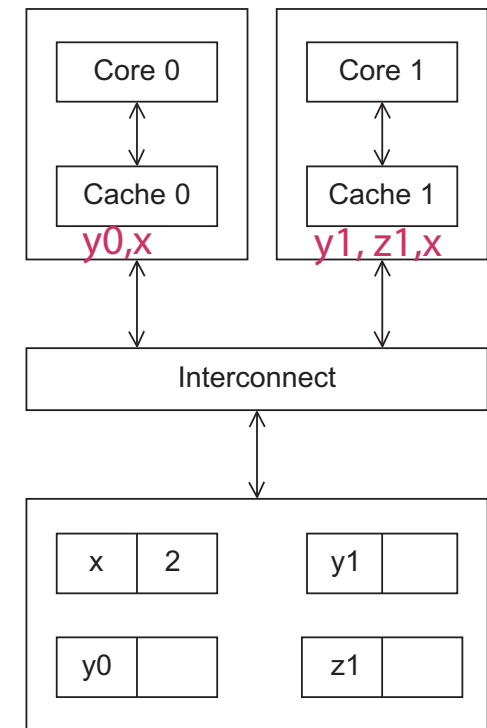


Figure 2.17

Cache Coherence Mechanisms

■ Snooping-base coherency

- The cores share a bus, i.e. any signal transmitted on the bus can be “seen” by all cores connected to the 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

■ Directory-based coherency

- Uses a data structure called a directory that stores the status of each cache line
- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated

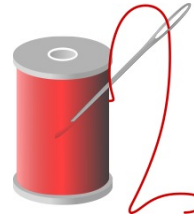
Parallel Software

- **Computer architectures are increasingly parallel**
 - not only in HPC but also in embedded computing
- **Parallel programming is still not pervasive in programming education but considered a “specialization”**
- **Compilers and tools have come a long way**
 - but fully automate parallelization is not a reality yet (and possibly will never be)
- **The burden to efficiently exploit parallel computing is on software**

SPMD – Single Program Multiple Data

- An 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;
```




Writing Parallel Programs

- 1. Divide the work among the processes/threads, such that**
 - each process/thread gets roughly the same amount of work
 - communication is minimized
- 2. Arrange for the processes/threads to synchronize**
- 3. Arrange for communication among processes/threads**

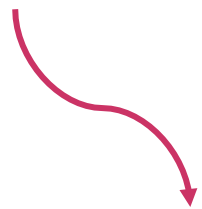
```
double x[n], y[n];  
...  
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

Nondeterminism (1)

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



Thread 1 > my_val = 19
Thread 0 > my_val = 7



Thread 0 > my_val = 7
Thread 1 > my_val = 19

order of execution in threads/processes non-deterministic, in this case not a huge problem

Nondeterminism (2)

```
my_val = Compute_val ( my_rank ) ;  
x += my_val ;
```

my_val, my_rank are private; x is shared

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val = 19 to x
5	Start other work	Store x = 19

order of threads makes a difference (race condition)

Nondeterminism (3)

- Race conditions can be avoided by protecting critical sections, e.g. with mutual exclusion locks (mutex)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```

Busy-Waiting

```
my_val = Compute_val ( my_rank ) ;  
if ( my_rank == 1 )  
    while ( !ok_for_1 ) ;    /* Busy-wait loop */  
x += my_val ;                /* Critical section */  
if ( my_rank == 0 )  
    ok_for_1 = true ;    /* Let thread 1 update x */
```

Message-Passing

```
char message [ 1 0 0 ] ;  
. . .  
my_rank = Get_rank ( ) ;  
if ( my_rank == 1) {  
    sprintf ( message , "Greetings from process 1" ) ;  
    Send ( message , MSG_CHAR , 100 , 0 ) ;  
} else if ( my_rank == 0) {  
    Receive ( message , MSG_CHAR , 100 , 1 ) ;  
    printf ( "Process 0 > Received: %s\n" , message ) ;  
}
```

Partitioned Global Address Space Languages

```
shared int n = . . . ;
shared double x [ n ] , y [ n ] ;
private int i, my_first_element , my_last_element ;
my_first_element = . . . ;
my_last_element = . . . ;
/ * Initialize x and y */
. . .
for (i = my_first_element ; i <= my_last_element ; i++)
    x [ i ] += y [ i ] ;
```


Input and Output

- **Parallel programs need specific rules how processes/threads can access I/O**
- **Standard input (stdin)**
 - in distributed memory programs, only process 0 will access stdin
 - in shared memory programs, only the master thread or thread 0 will access stdin
- **Standard output (stdout) and standard error (stderr)**
 - in distributed memory and shared memory programs all the processes/threads can access stdout and stderr
 - because of the non-determinism of the order of output to stdout only a single process/thread should be used for all output to stdout other than debugging output
 - debug output should include the rank/id of the process/thread generating the output
- **File I/O**
 - only a single process/thread shall access any single file (other than stdin, stdout, or stderr)
 - each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

Performance



Speedup of a Parallel Program

- Number of cores = n
- Serial run-time = T_{serial}
- Parallel run-time = T_{parallel}
- Speedup S
- Efficiency E

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

$$T_{\text{parallel}} = T_{\text{serial}}/n \quad \text{linear speedup}$$

$$E = \frac{S}{n} = \frac{\frac{T_{\text{serial}}}{T_{\text{parallel}}}}{n} = \frac{T_{\text{serial}}}{n \cdot T_{\text{parallel}}}$$

Example: Speedups and Efficiencies

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

Amdahl's Law

- **Gene Amdahl's seminal paper from 1967 makes the case that speeding up programs with parallel processing is severely limited**

- only a fraction p of the each application will be parallelizable
- even with perfect speedup for the parallelizable part, the overall achievable speedup will be dominated by the computation time of the remaining (serial) part

- **Model**

- p : percentage of execution time that benefits from parallelization
- $1-p$: percentage of the execution time not benefitting from parallelization (serial part)
- n : speedup of the parallelizable part (= number of processors assuming linear speedup)
- T : execution time of the program running on 1 processor

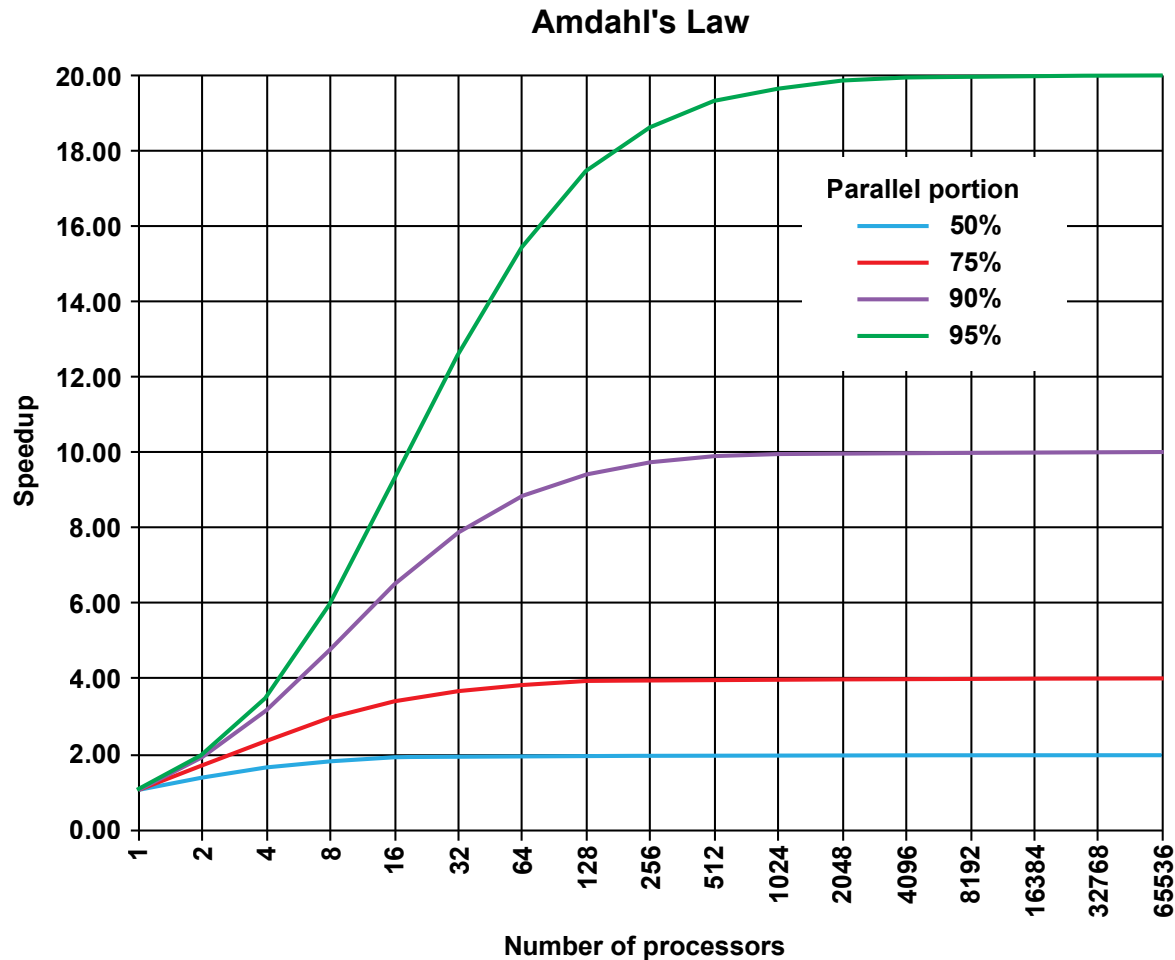
$$S(n) = \frac{T_{serial}}{T_{parallel}} = \frac{T}{(1-p)T + \frac{T}{n}} = \frac{1}{(1-p) + \frac{1}{n}}$$

$$\lim_{n \rightarrow \infty} S(n) = \frac{1}{1-p}$$

G. M. Amdahl. **Validity of the single processor approach to achieving large scale computing capabilities.**

In *Proc. Spring Joint Computer Conference (SJCC)*, pages 483–485, New York, 1967. ACM.

Amdahl's Law (2)



Gustafson-Barsis' Law

- **Amdahl's law suggests that parallel computing “does not work”**
 - limited speedup and decreasing efficiency with increasing number of processors
 - still, parallel computing is successfully used and many codes run with high efficiency
- **Gustafson writes seminal paper in 1988 arguing for measuring speedup by scaling the problem size instead of time**
 - Amdahl assumes that the problem size is constant, use more processor to reduce time (**strong scaling**)
 - Gustafson assumes that time to solution is constant, use more processor to solve larger problems (**weak scaling**)

Gustafson-Barsis' Law (2)

■ Model

- t_s (t_p) computation time for sequential (parallel) part
- n : number of processors
- w_p : workload on parallel system $w_p = t_s + t_p$
- w_1 : workload on sequential system $w_1 = t_s + N \cdot t_p$

$$S(n) = \frac{w_1}{w_2} = \frac{t_s + n \cdot t_p}{t_s + t_p}$$

let $f^* = \frac{t_s}{t_s + t_p}$ percentage of sequential
computation on the parallel system

$$S(n) = f^* + n \cdot (1 - f^*)$$

Example: Speedups and Efficiencies of Parallel Program on Different Problem Sizes

	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

Speedup

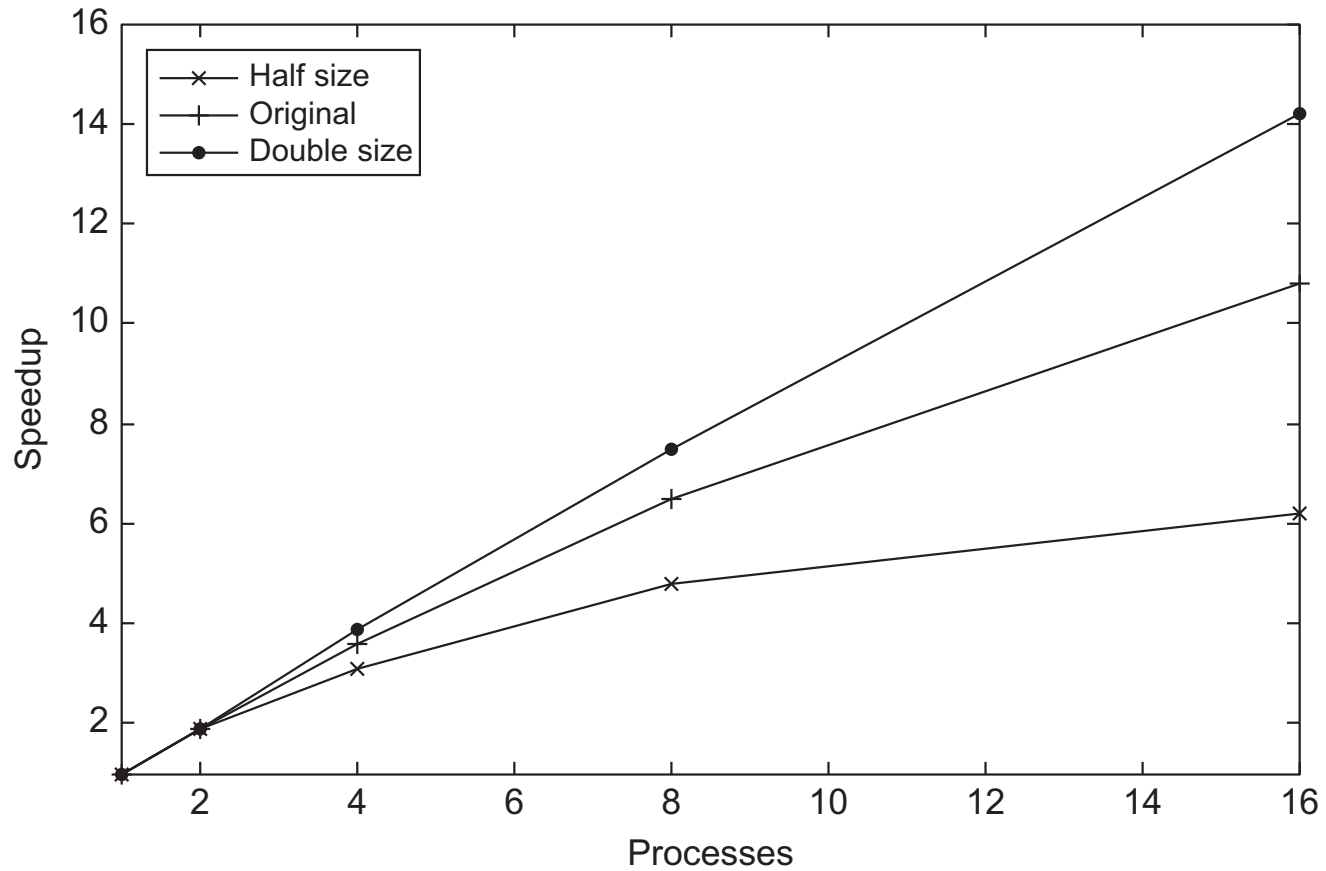


Figure 2.18

Efficiency

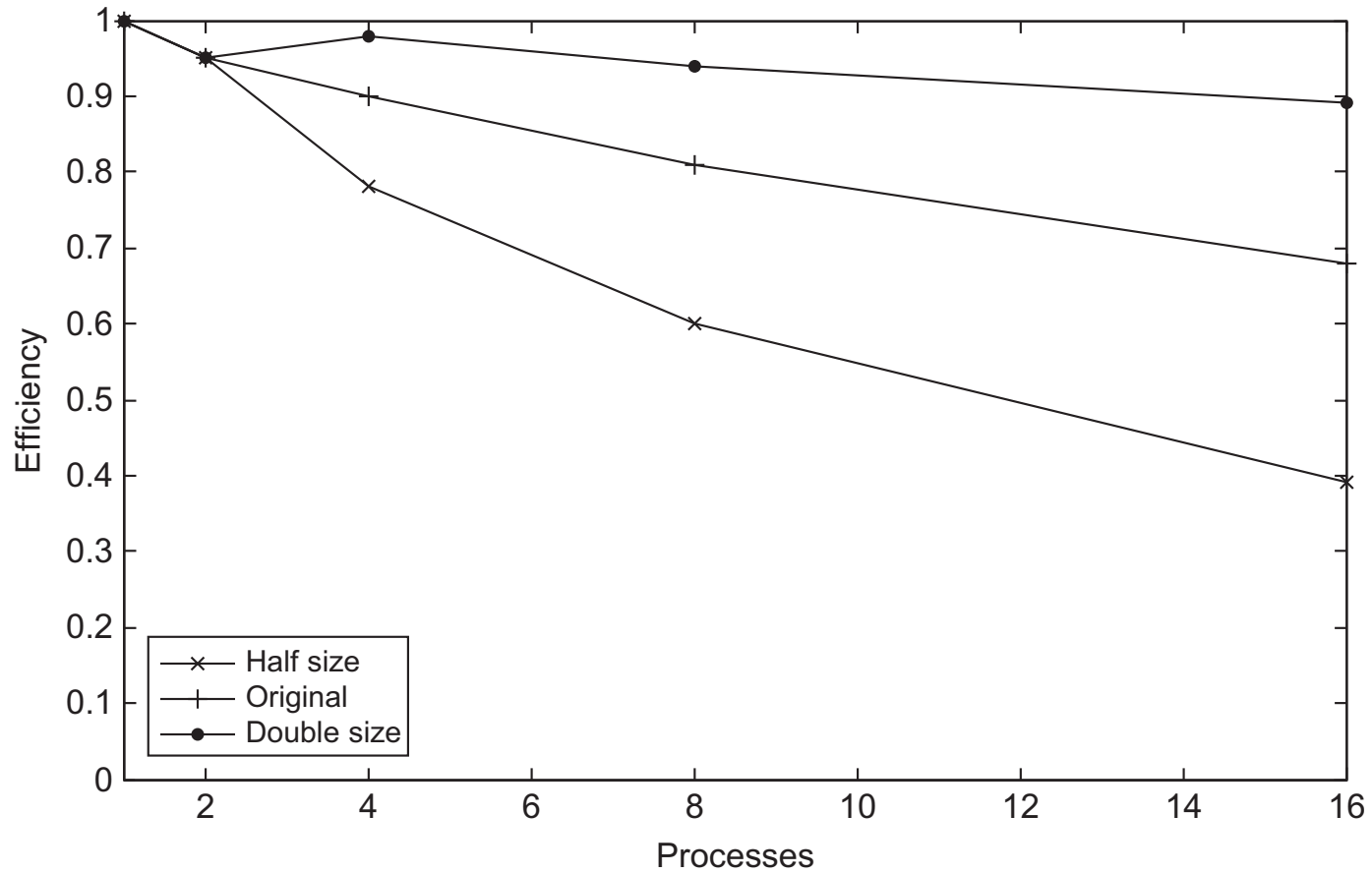
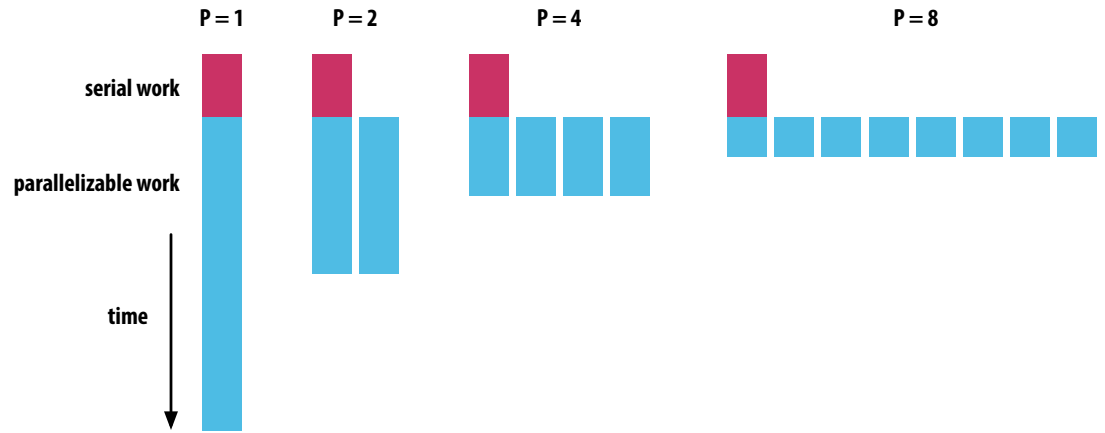


Figure 2.19

Amdahl's vs. Gustafson-Barsis' Law Illustrated

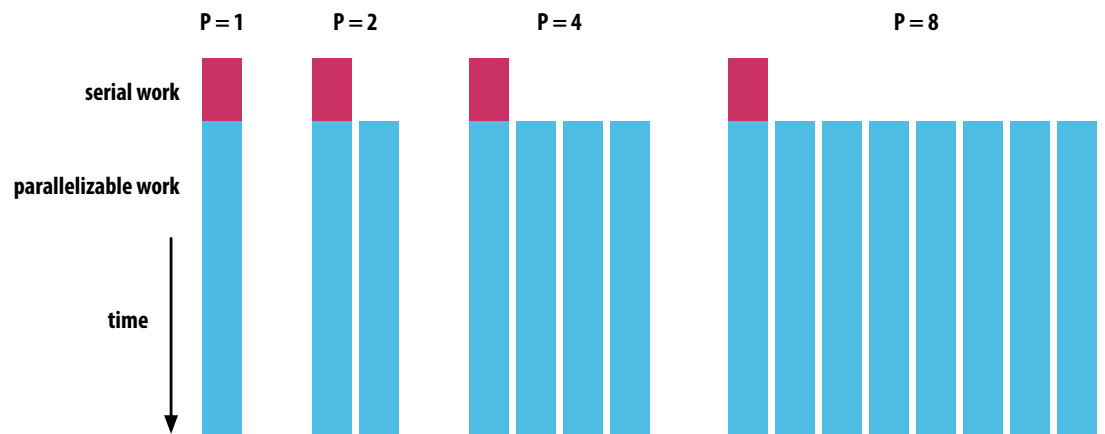
Amdahl

- figure of merit: execution time
- total workload constant
- speedup limited
- strong scaling



Gustafson-Barsis

- figure of merit: workload
- time to execution constant
- speedup unlimited
- weak scaling



Scalability

- In general, a problem is scalable if it can handle ever increasing problem sizes
- If we can increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is **strongly scalable**
- If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is **weakly scalable**

Taking Timings

- **What is time?**
- **Start to finish?**
- **A program segment of interest?**
- **CPU time?**
- **Wall clock time?**

Taking Timings

**theoretical
function**

```
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);
```

MPI_Wtime

omp_get_wtime

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);
```


Taking Timings

```
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);
```

Foster's methodology for Parallel Program Design

■ Step 1 – Partitioning

- divide the computation to be performed and the data operated on by the computation into small tasks
- keep the focus on identifying tasks that can be executed in parallel

■ Step 2 – Communication

- determine what communication needs to be carried out among the tasks identified in the previous step

■ Step 3 – Agglomeration or aggregation

- combine tasks and communications identified in the first step into larger tasks
- e.g., if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task

■ Step 4 – Mapping

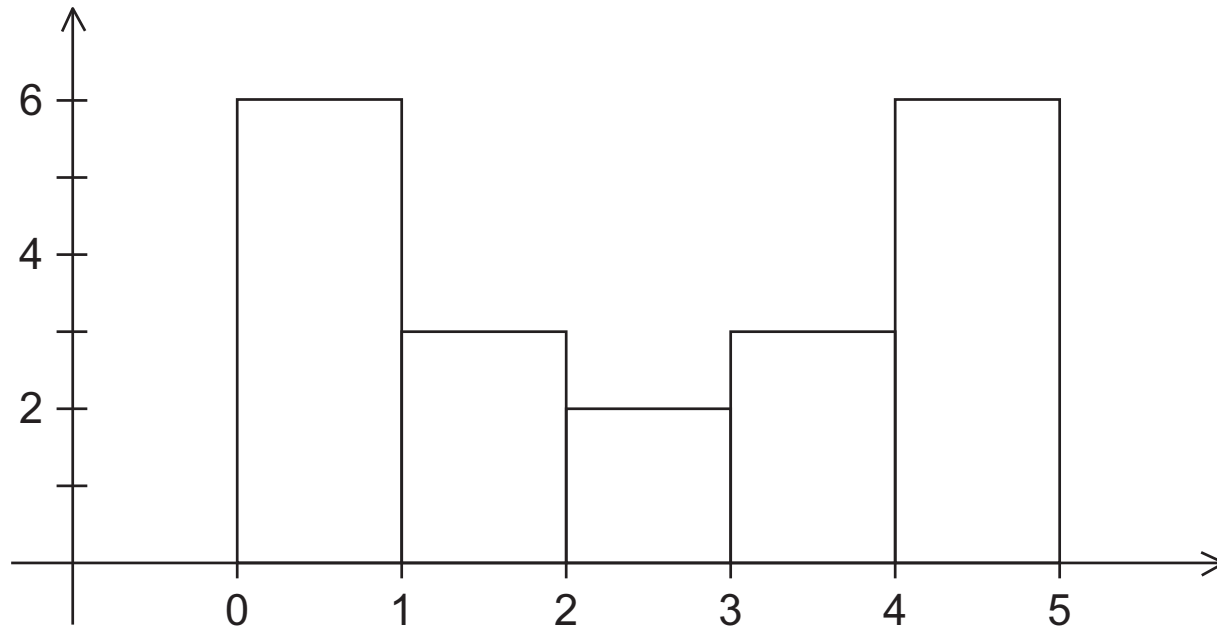
- assign the composite tasks identified in the previous step to processes/threads
- strive for minimizing communication
- each process/thread should get roughly the same amount of work

Example – Histogram Computation

raw data

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2,

0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



histogram

Serial program

■ Input

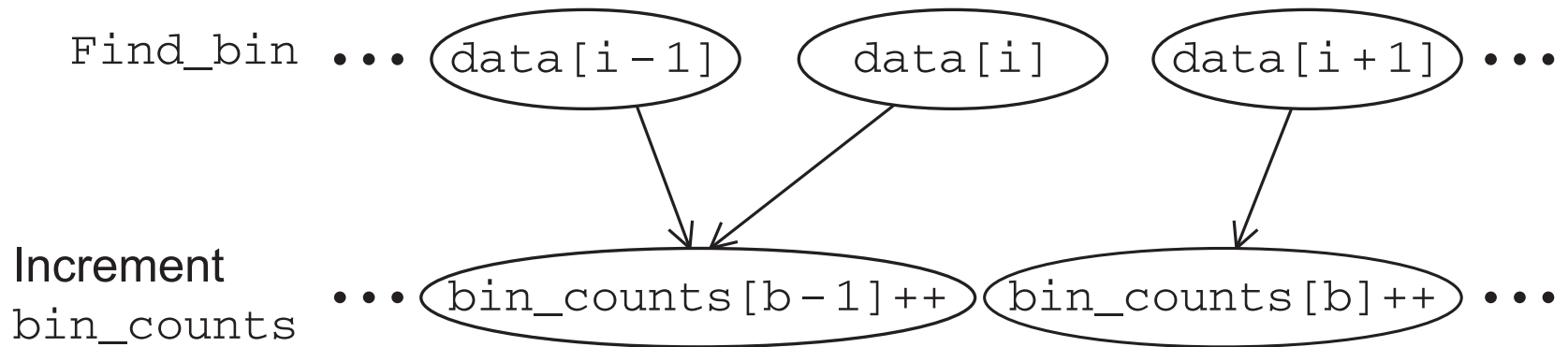
- number of measurements: `data_count`
- array of `data_count` floats: `data`
- minimum value for bin containing the smallest values: `min_meas`
- maximum value for the bin containing the largest values: `max_meas`
- number of bins: `bin_count`

■ Output

- `bin_maxes`: an array of `bin_count` floats
- `bin_counts`: an array of `bin_count` ints

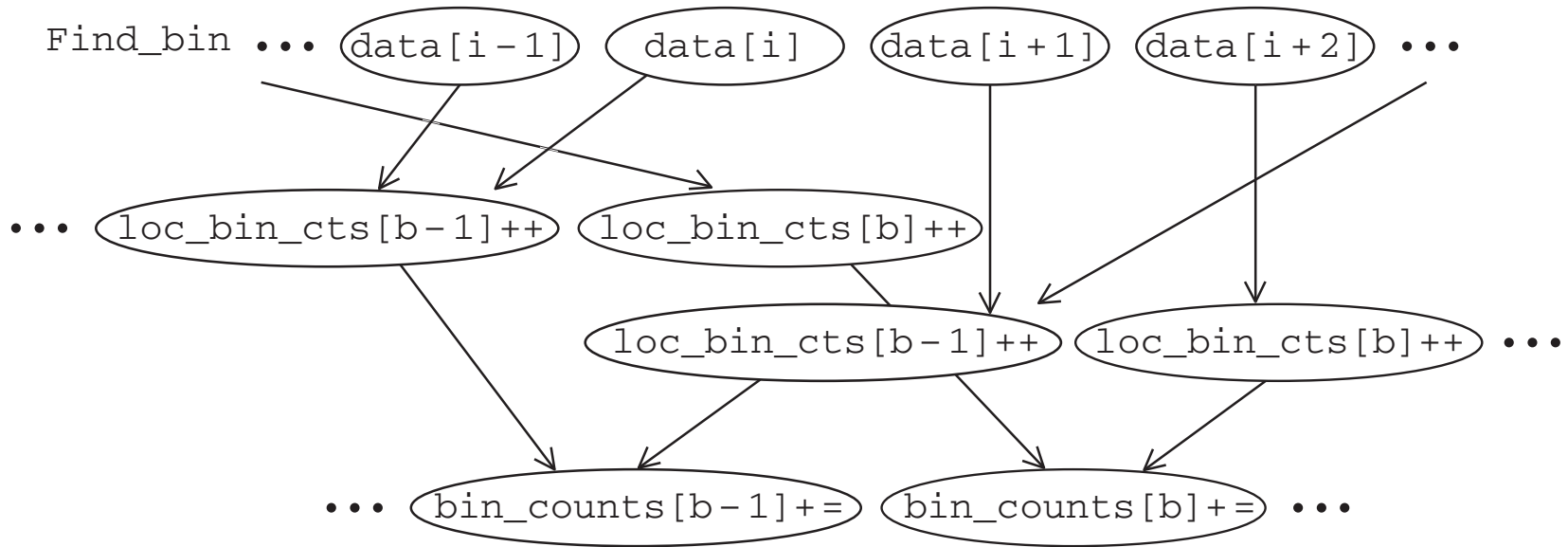
First two stages of Foster's Methodology

- Partitioning
- Communication



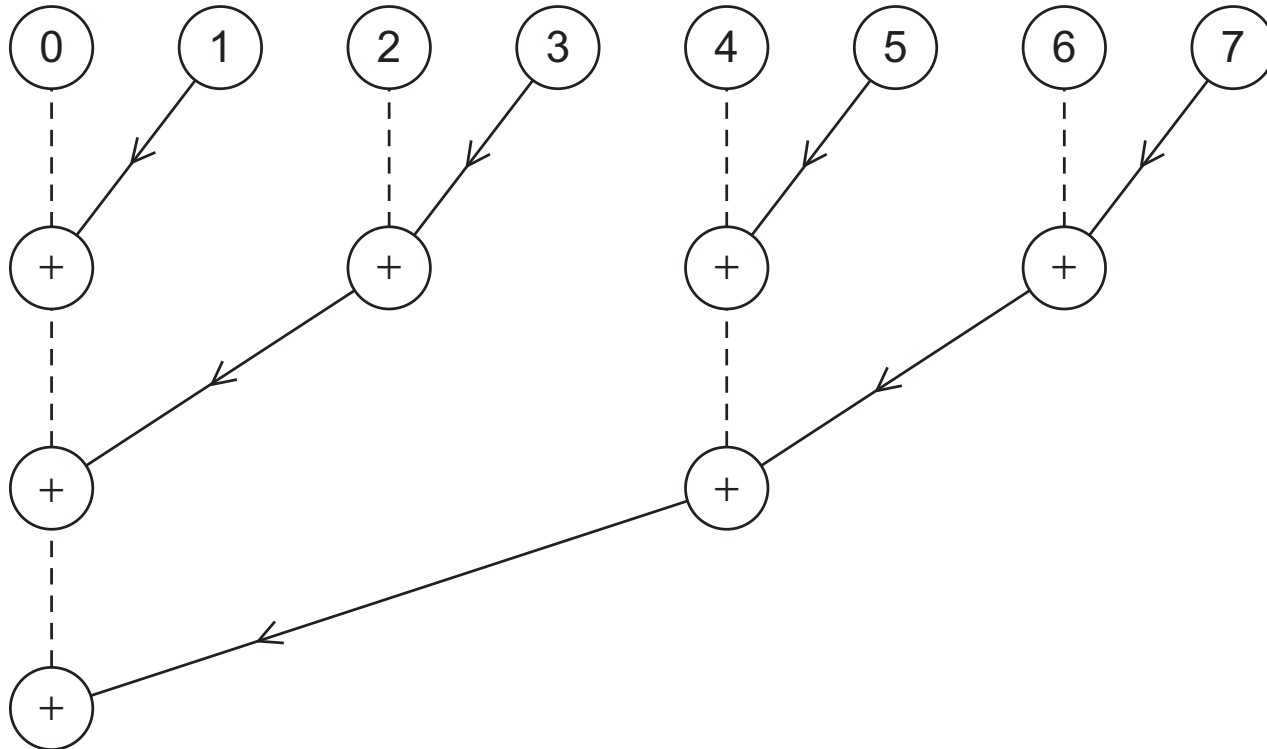
lots of potential parallelism, but conflicts when incrementing bin_counts

Alternative Definition of Tasks and Communication



aggregate local counts for smaller batches of data first, reduce conflicts for updates of bin_counts

Adding the Local Arrays



Concluding Remarks (1)

■ **Serial systems**

- The standard model of computer hardware has been the von Neumann architecture

■ **Parallel hardware**

- Flynn's taxonomy

■ **Parallel software**

- We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching
- SPMD programs

■ **Input and Output**

- We'll write programs in which 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

Concluding Remarks (2)

■ Performance

- Speedup
- Efficiency
- Amdahl's law
- Gustafson-Barsis' law
- Scalability

■ Parallel Program Design

- Foster's methodology

Acknowledgements

- **Peter S. Pacheco / Elsevier**

- for providing the lecture slides on which this presentation is based

Change log

■ 1.1.1 (2017-10-16)

- clarify slide 13, 46, 79, 120, 121
- merge slide 35 + 36, 70+71
- removed slide 75 (dynamic vs. static threads)

■ 1.1.0 (2017-10-13)

- updated slides for winter term 2017/18
- correction of minor typos, cosmetics

Change log

- **1.0.4 (2016-11-11)**
 - fix typo on slide 69
- **1.0.3 (2016-11-10)**
 - add illustration Amdahl vs. Gustafson-Barsis
- **1.0.2 (2016-11-09)**
 - finalize slides for second part
- **1.0.1 (2016-11-04)**
 - fix typo in code on slide 22
 - numerous cosmetic changes, improve legibility of tables and figures
- **1.0.0 (2016-11-04)**
 - initial version of slides