IF3130
Sistem Paralel dan Terdistribusi
Parallel Hardware and Parallel Software

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Robithoh Annur (robithoh@stafff.stei.itb.ac.id)
Andreas Bara Timur (bara@staff.stei.itb.ac.id)

Content

Instruction Level Parallelism (ILP)

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

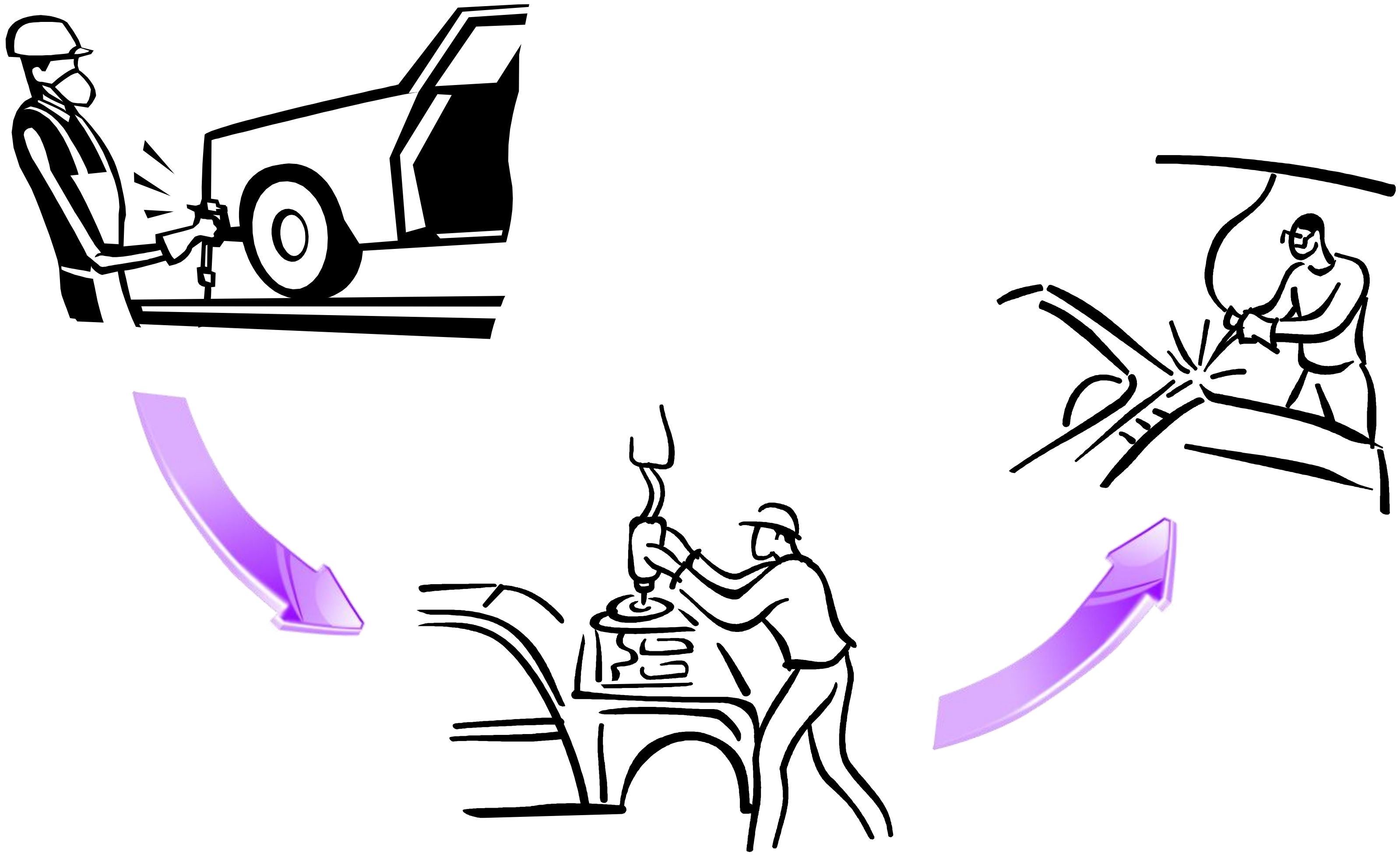
Copyright © 2010, Elsevier Inc. All rights Reserved

Instruction Level Parallelism (2)

- **Pipelining** - functional units are arranged in stages.
- **Multiple issue** - multiple instructions can be simultaneously initiated.

Copyright © 2010, Elsevier Inc. All rights Reserved

Pipelining



Copyright © 2010, Elsevier Inc. All rights Reserved

Pipelining example (1)

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

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Pipelining example (2)

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

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

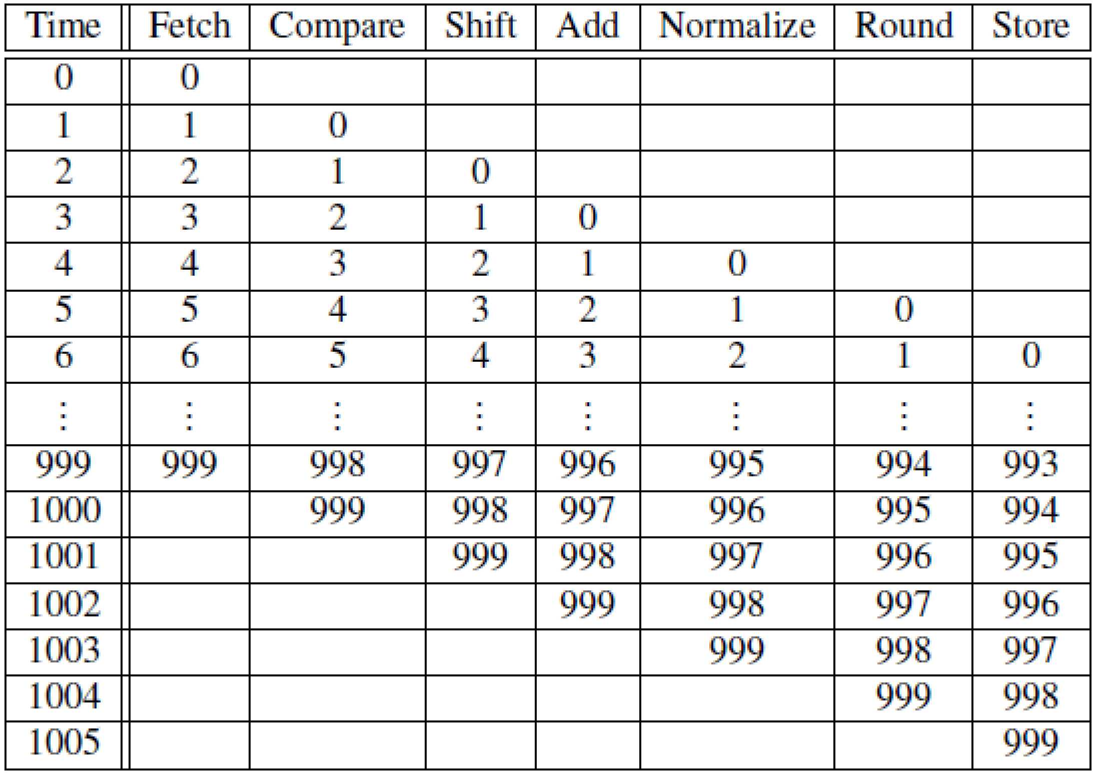
Copyright © 2010, Elsevier Inc. All rights Reserved

Pipelining (3)

- 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.

Copyright © 2010, Elsevier Inc.
All rights Reserved

Pipelining (4)



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.

Copyright © 2010, Elsevier Inc.
All rights Reserved

Pipelining (5)

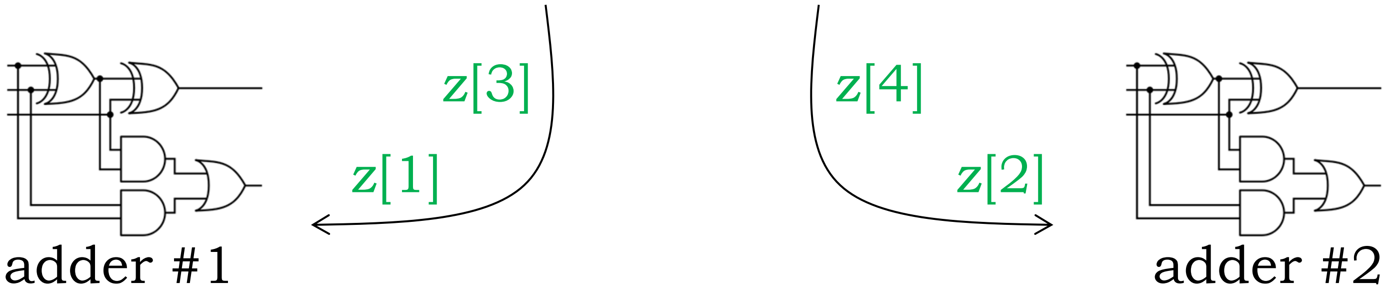
- One floating point addition still takes 7 nanoseconds.
- But 1000 floating point additions now takes 1006 nanoseconds!

Copyright © 2010, Elsevier Inc. All rights Reserved

Multiple Issue (1)

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

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



Copyright © 2010, Elsevier Inc. All rights Reserved

Multiple Issue (2)

- **static** multiple issue - functional units are scheduled at compile time.

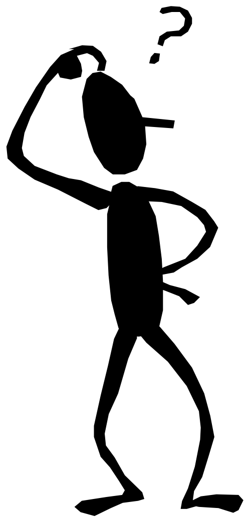
- **dynamic** multiple issue – functional units are scheduled at run-time.

superscalar

Copyright © 2010, Elsevier Inc. All rights Reserved

Speculation (1)

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.



- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

Copyright © 2010, Elsevier Inc. All rights Reserved

Speculation (2)

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

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Hardware multithreading (1)

- There aren’t always good opportunities for simultaneous execution of different threads.
- Hardware multithreading provides a means for systems 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.

Copyright © 2010, Elsevier Inc. All rights Reserved

Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.
  - Pros: potential to avoid wasted machine time due to stalls.
  - Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

Copyright © 2010, Elsevier Inc. All rights Reserved

Hardware multithreading (3)

- **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.

Copyright © 2010, Elsevier Inc. All rights Reserved

Hardware multithreading (3)

- **Simultaneous multithreading** (SMT) - a variation on fine-grained multithreading.

- Allows multiple threads to make use of the multiple functional units.

Copyright © 2010, Elsevier Inc. All rights Reserved

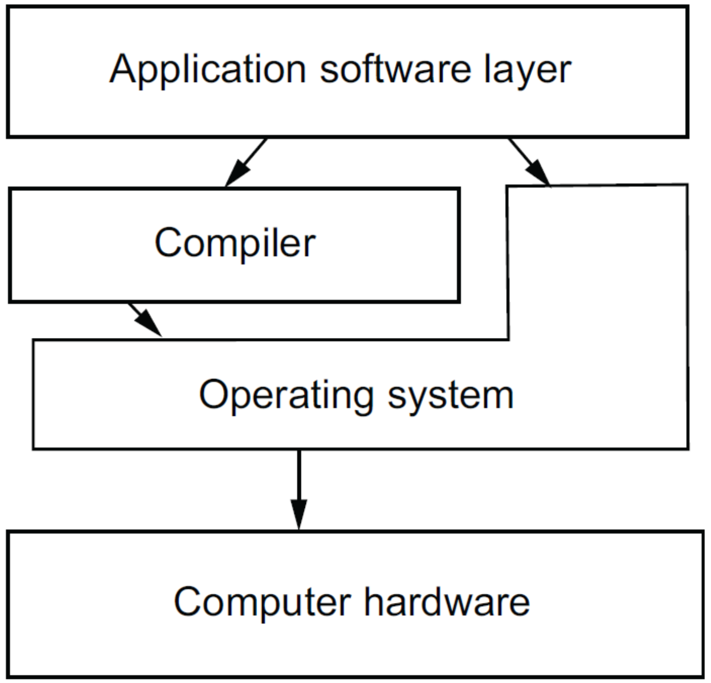
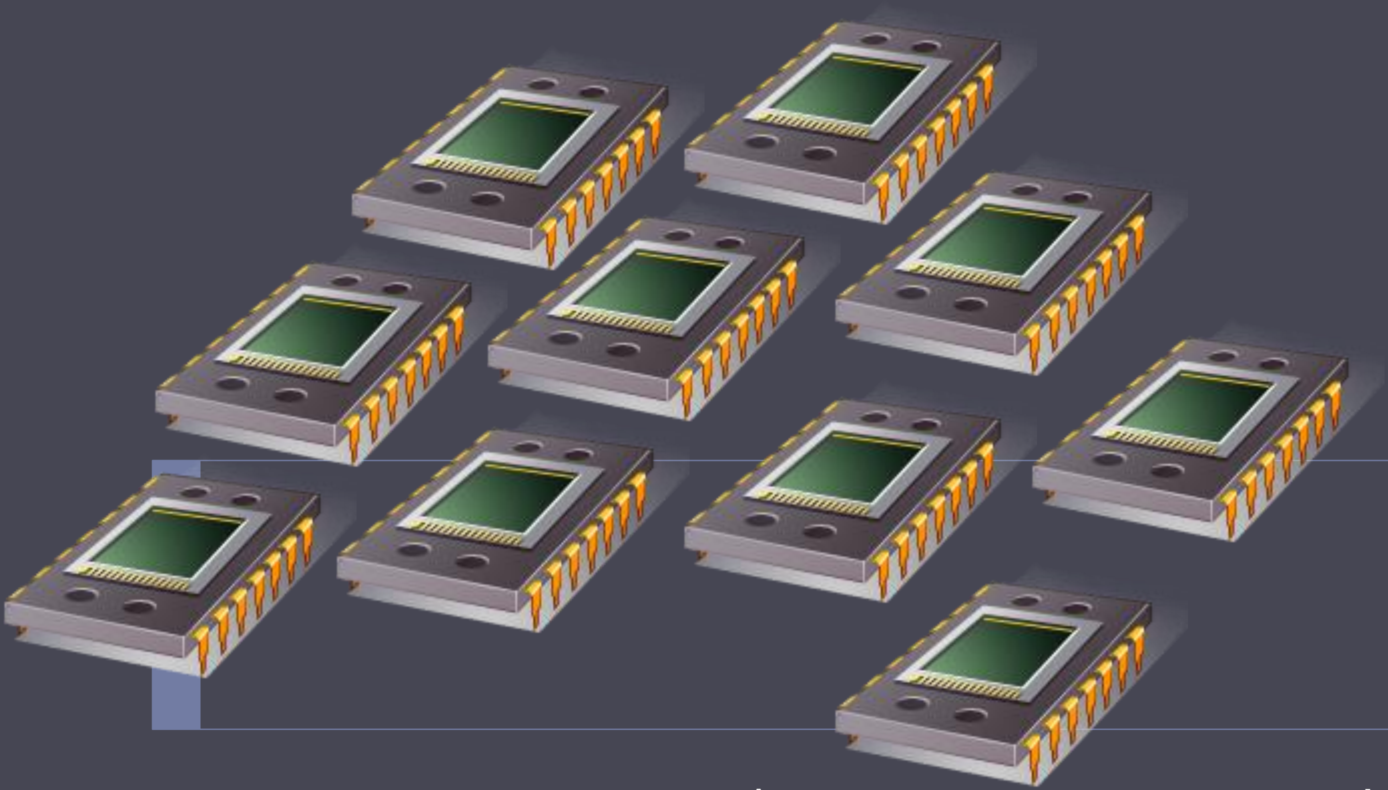


Figure 1.7 Parallelization is expressed in an application software layer that gets mapped to the computer hardware through the compiler and the OS.



A programmer can write code to exploit.

Parallel hardware

Copyright © 2010, Elsevier Inc. All rights

Flynn's Taxonomy

classic von Neumann

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

not covered

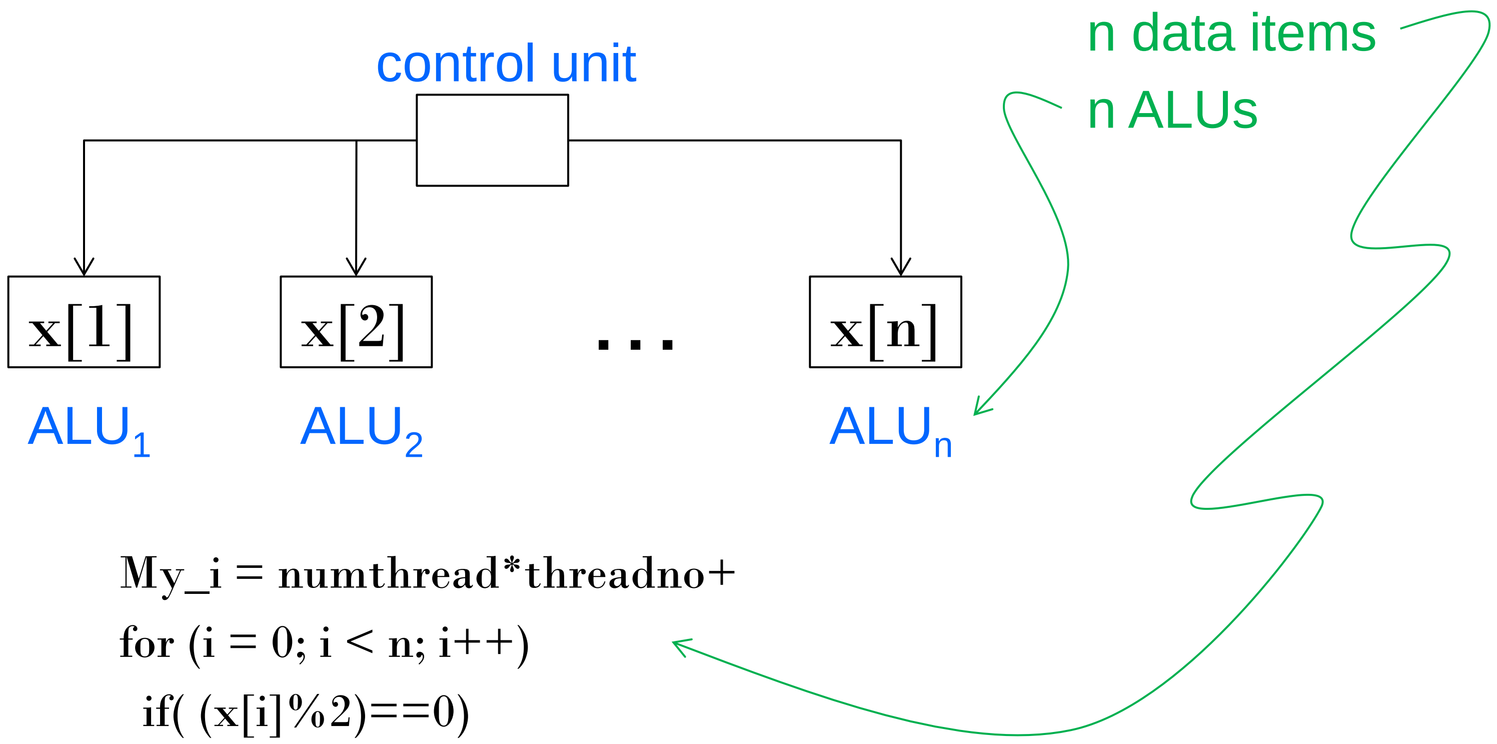
Copyright © 2010, Elsevier Inc. All rights Reserved

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 example



```
My_i = numthread*threadno+
for (i = 0; i < n; i++)
  if( (x[i]%2)==0)
       x[i] += y[i];
   else
    x[i] -= y[i];
```

Copyright © 2010, Elsevier Inc. All rights Reserved

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.

Round3	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 other types of more complex parallel problems.

Copyright © 2010, Elsevier Inc. All rights Reserved

Vector processors (1)

- Operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars.

- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.

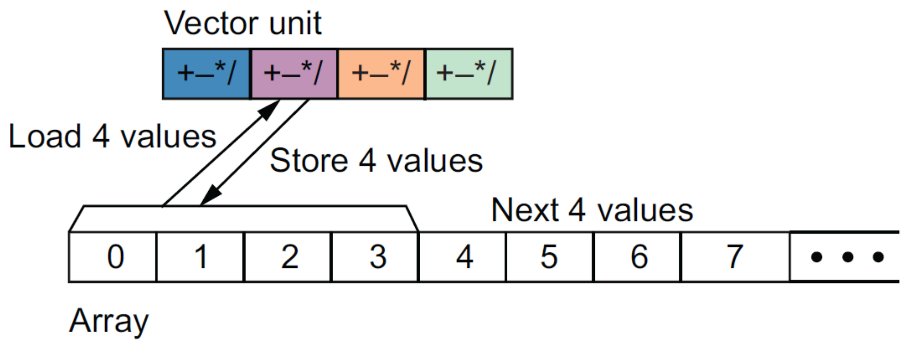


Figure 1.17 Vector processing example with four array elements operated on simultaneously

Copyright © 2010, Elsevier Inc.
All rights Reserved

Vector processors (2)

- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector (or pairs of elements).
- Vector instructions.
  - Operate on vectors rather than scalars.

Copyright © 2010, Elsevier Inc. All rights Reserved

Vector processors (3)

- Interleaved memory.
  - Multiple "banks" of memory, which can be accessed more or less independently.
  - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather.
  - The program accesses elements of a vector located at fixed intervals.

Copyright © 2010, Elsevier Inc. All rights Reserved

Vector processors - 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.
- Uses every item in a cache line.

Copyright © 2010, Elsevier Inc. All rights Reserved

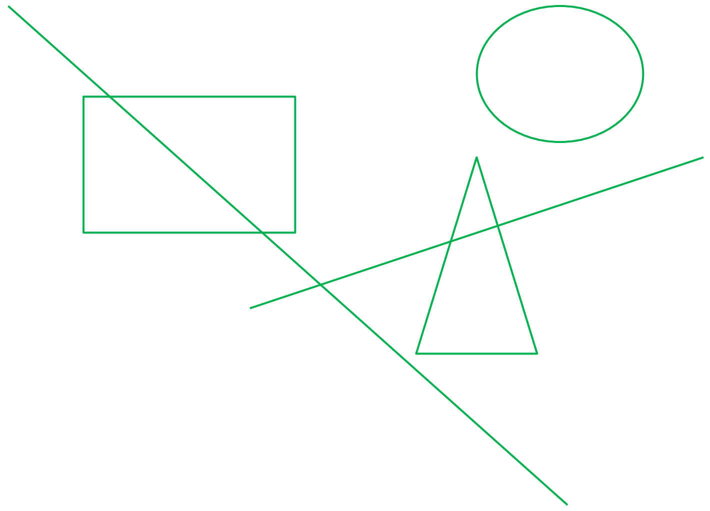
Vector processors - Cons

- They don't handle irregular data structures as well as other parallel architectures.
- A very finite limit to their ability to handle ever 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.



Copyright © 2010, Elsevier Inc. All rights Reserved

GPUs

- 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.

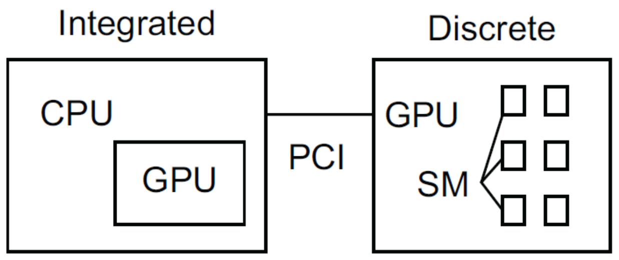


Figure 1.18 GPUs come in two varieties: integrated and discrete. Discrete or dedicated GPUs typically have a large number of streaming multiprocessors and their own DRAM. Accessing data on a discrete GPU requires communication over a PCI bus.

Copyright © 2010, Elsevier Inc. All rights Reserved

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.

Copyright © 2010, Elsevier Inc. All rights Reserved

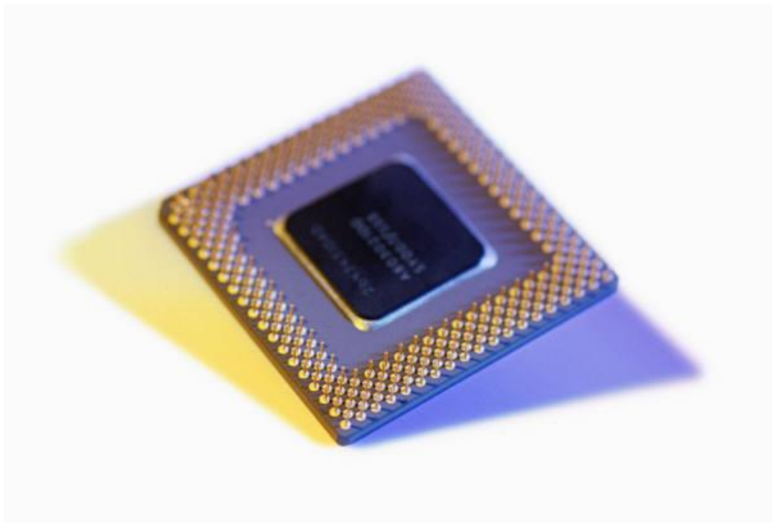
Shared Memory System (1)

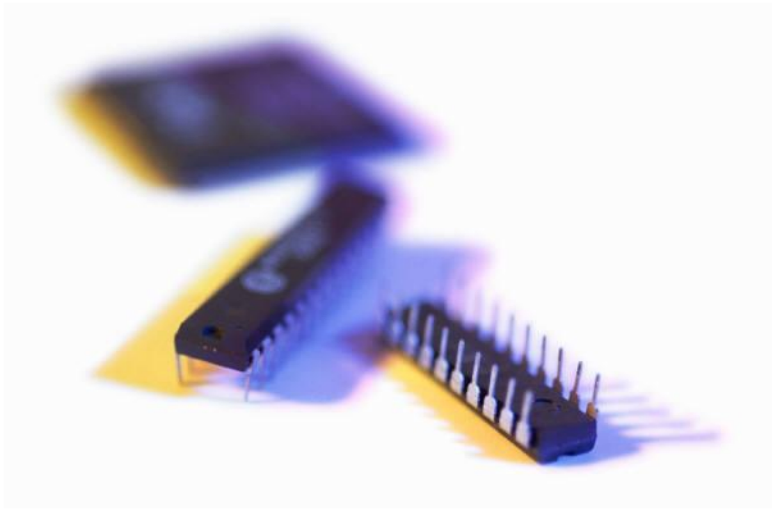
- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

Copyright © 2010, Elsevier Inc. All rights Reserved

Shared Memory System (2)

- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU's or cores on a single chip)





Copyright © 2010, Elsevier Inc. All rights Reserved

Shared Memory System

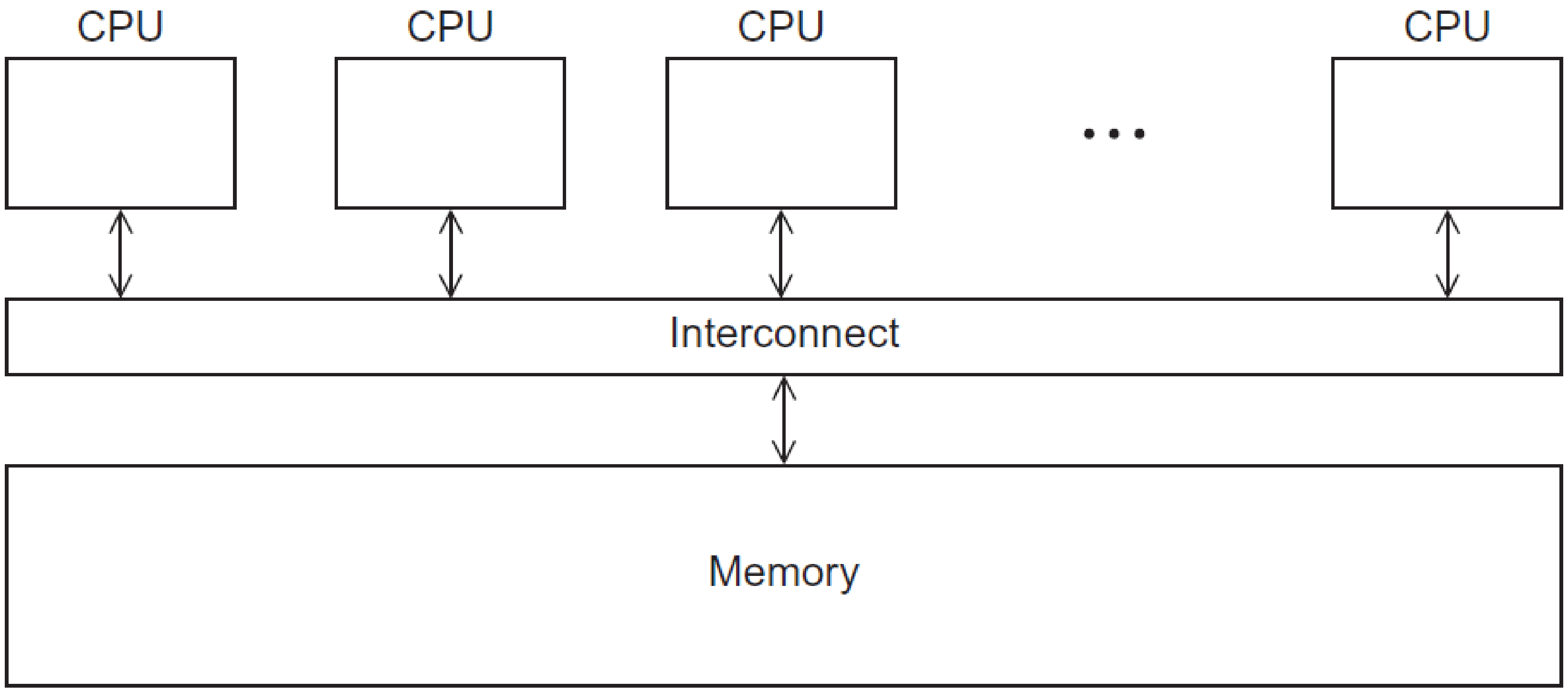
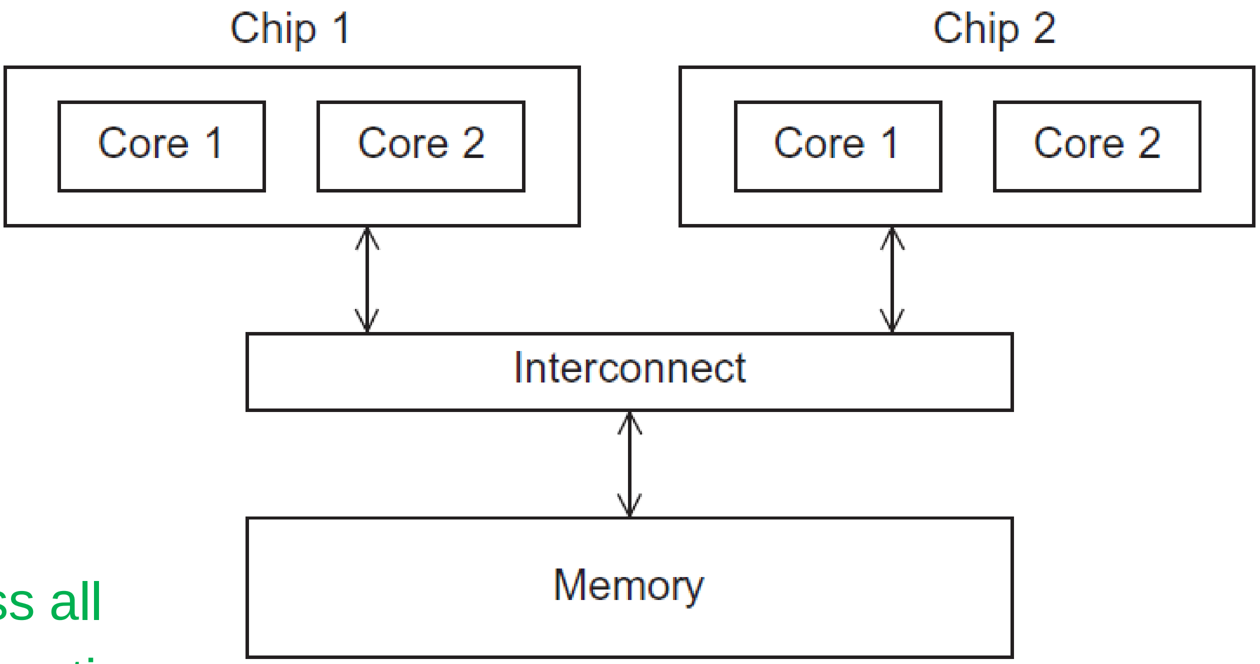


Figure 2.3

Copyright © 2010, Elsevier Inc.
All rights Reserved

UMA multicore system

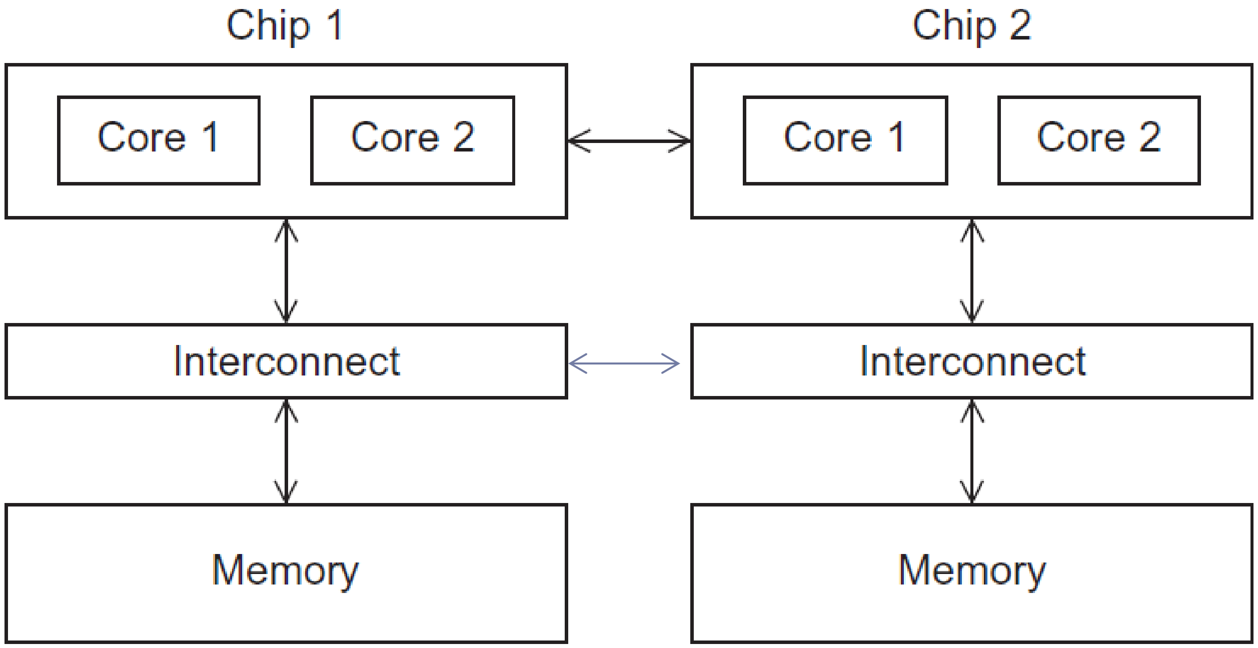


Time to access all the memory locations will be the same for all the cores.

Figure 2.5

Copyright © 2010, Elsevier Inc. All rights Reserved

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.
- Nodes of a cluster are individual computations units joined by a communication network.

a.k.a. hybrid systems

Copyright © 2010, Elsevier Inc. All rights Reserved

Distributed Memory System

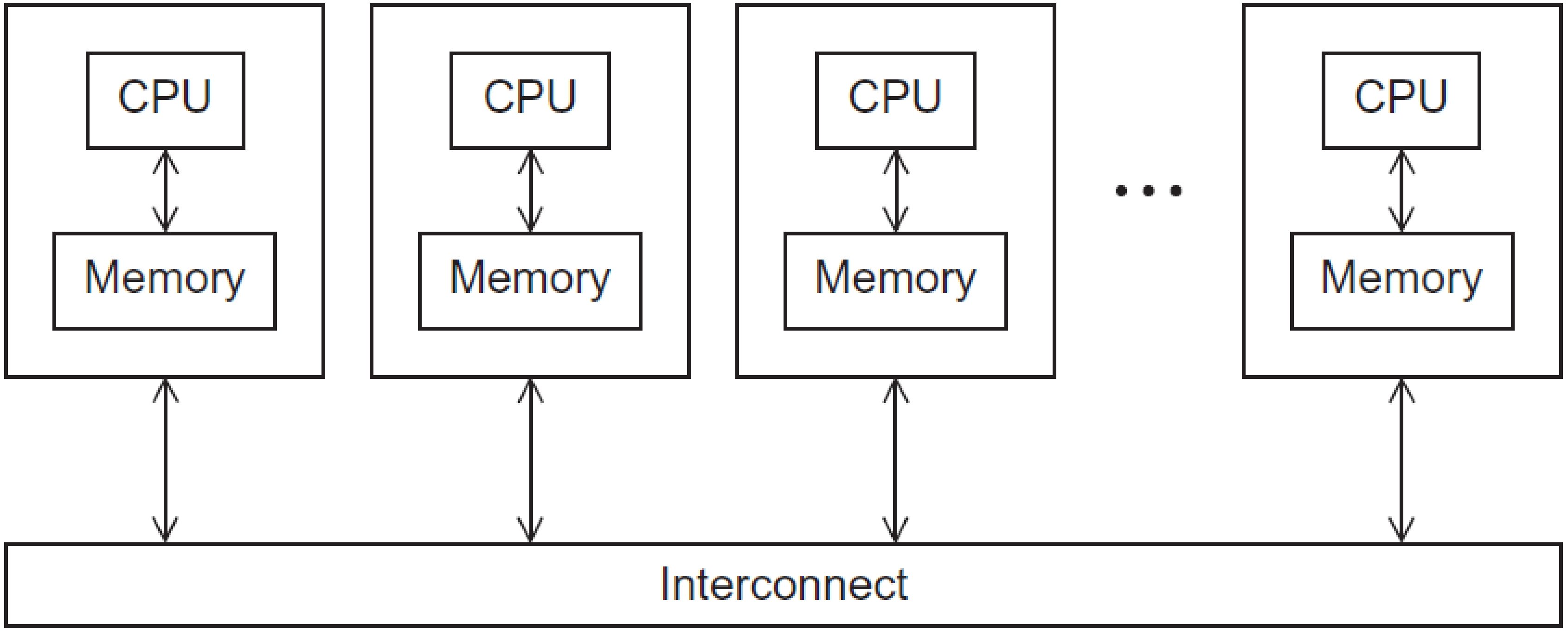


Figure 2.4

Copyright © 2010, Elsevier Inc. All rights Reserved

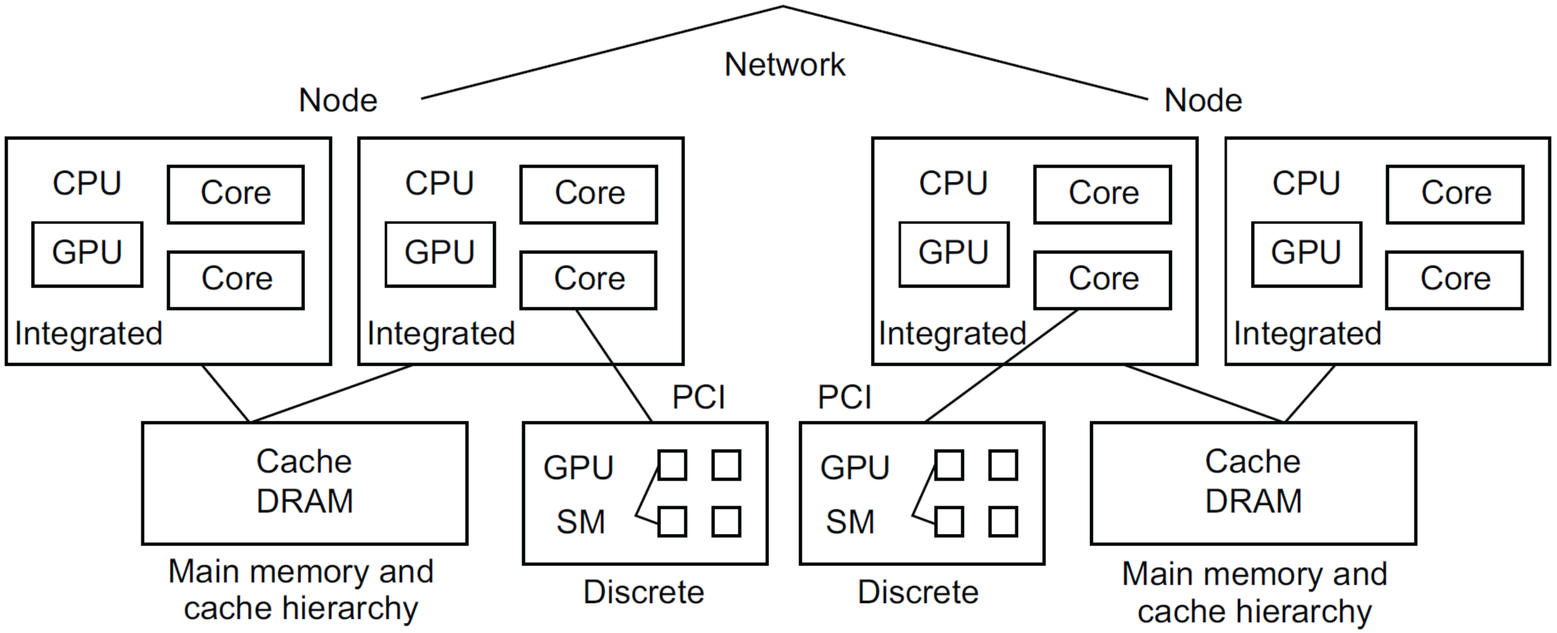


Figure 1.19 A general heterogeneous parallel architecture model consisting of two nodes connected by a network. Each node has a multi-core CPU with an integrated and discrete GPU and some memory (DRAM). Modern compute hardware normally has some arrangement of these components.

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 interconnect
  - A collection of parallel communication wires together with some hardware that controls access to the bus.
  - Communication wires are shared by the devices that are connected to it.
  - As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

Copyright © 2010, Elsevier Inc. All rights Reserved

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 the cost of the switches and links is relatively high.

Copyright © 2010, Elsevier Inc. All rights Reserved

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

Bisection width

- A measure of "number of simultaneous communications" or "connectivity".
- How many simultaneous communications can take place "across the divide" between the halves?

Copyright © 2010, Elsevier Inc. All rights Reserved

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.

Copyright © 2010, Elsevier Inc. All rights Reserved

More definitions

- 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.
- 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

Message transmission time = l + n / b

latency (seconds)

length of message (bytes)

bandwidth (bytes per second)

Copyright © 2010, Elsevier Inc.
All rights Reserved

Cache coherence

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

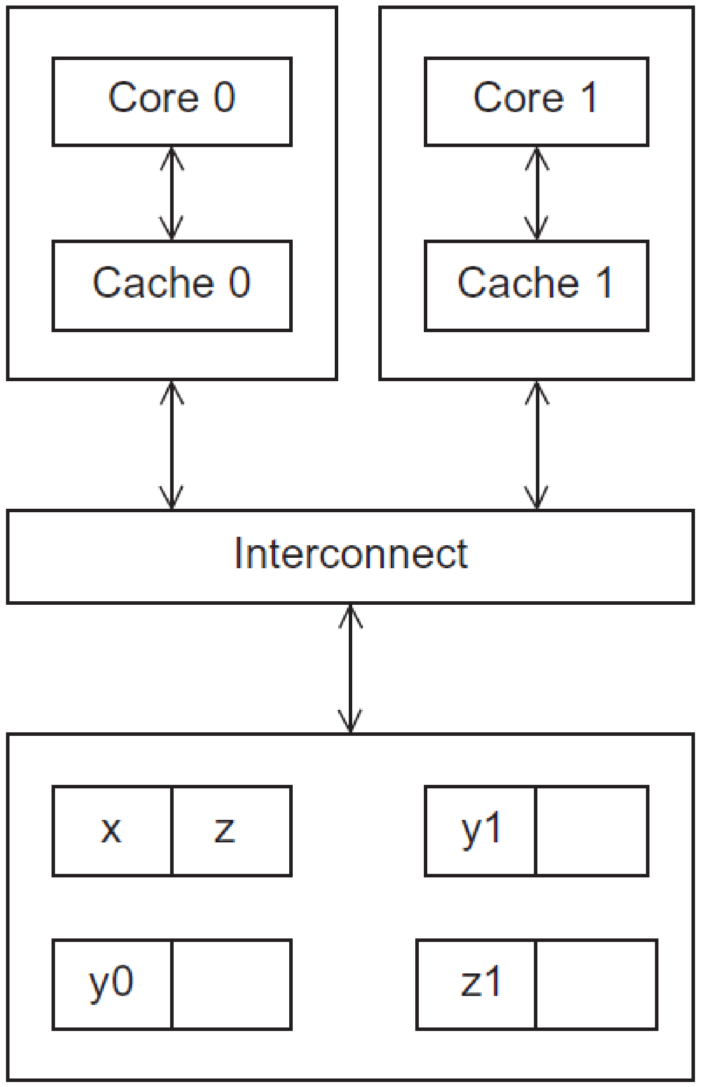


Figure 2.17

A shared memory system with two cores and two caches

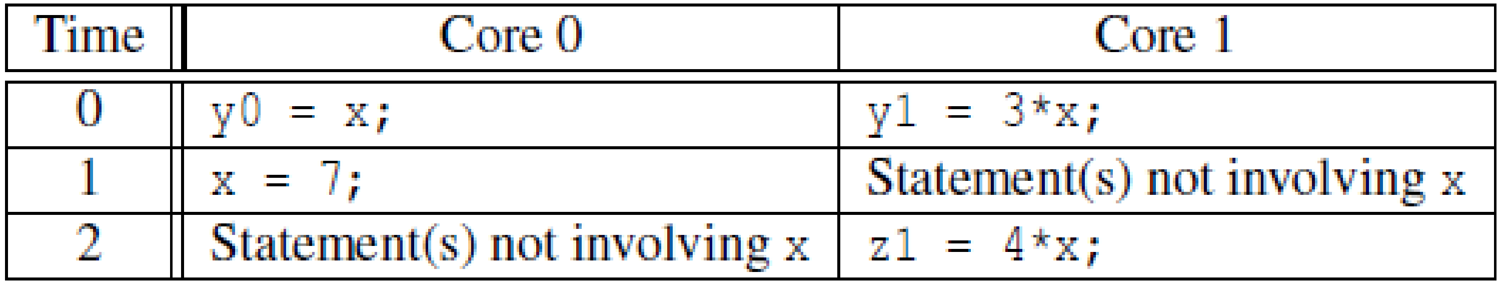
Copyright © 2010, Elsevier Inc.
All rights Reserved

Cache coherence

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 = ???

Copyright © 2010, Elsevier Inc.
All rights Reserved

Snooping Cache Coherence

- The cores share a bus .
- 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.

Copyright © 2010, Elsevier Inc.
All rights Reserved

Directory Based Cache Coherence

- 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.

Copyright © 2010, Elsevier Inc. All rights Reserved

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

Common techniques for parallelization

- Process-based parallelization – Message passing
- Thread-based parallelization – Shared data via memory
- Vectorization – multiple operations with one instruction
- Stream processing – through specialized processors

Process based parallelization

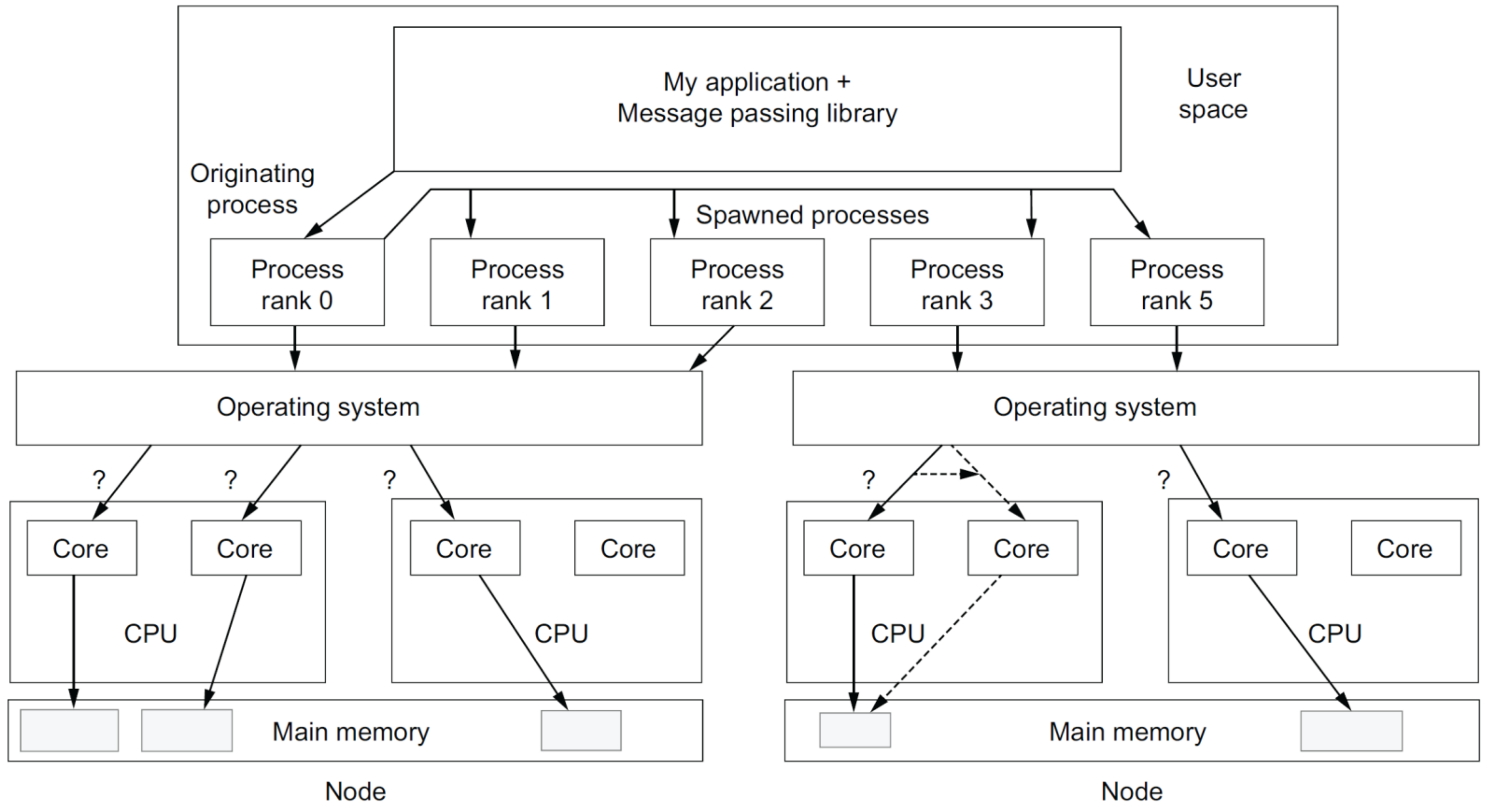


Figure 1.20 The message passing library spawns processes. The OS places the processes on the cores of two nodes. The question marks indicate that the OS controls the placement of the processes and can move these during run time as indicated by the dashed arrows. The OS also allocates memory for each process from the node's main memory.

Thread-based parallelization

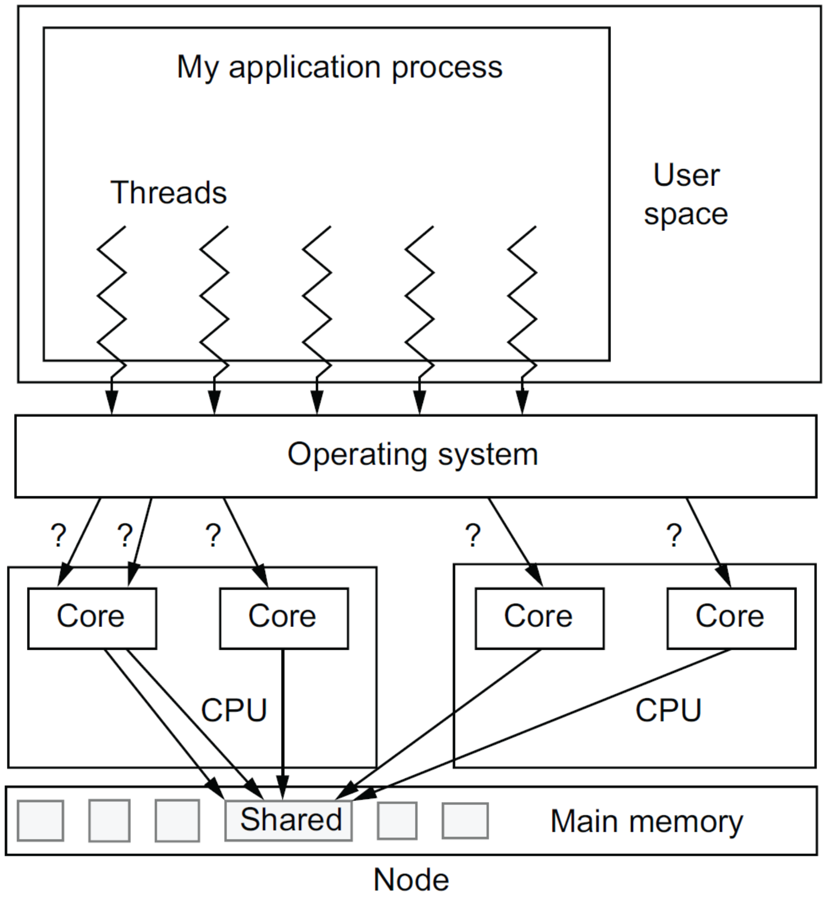


Figure 1.21 The application process in a thread-based approach to parallelization spawns threads. The threads are restricted to the node's domain. The question marks show that the OS decides where to place the threads. Some memory is shared between threads.

vectorization

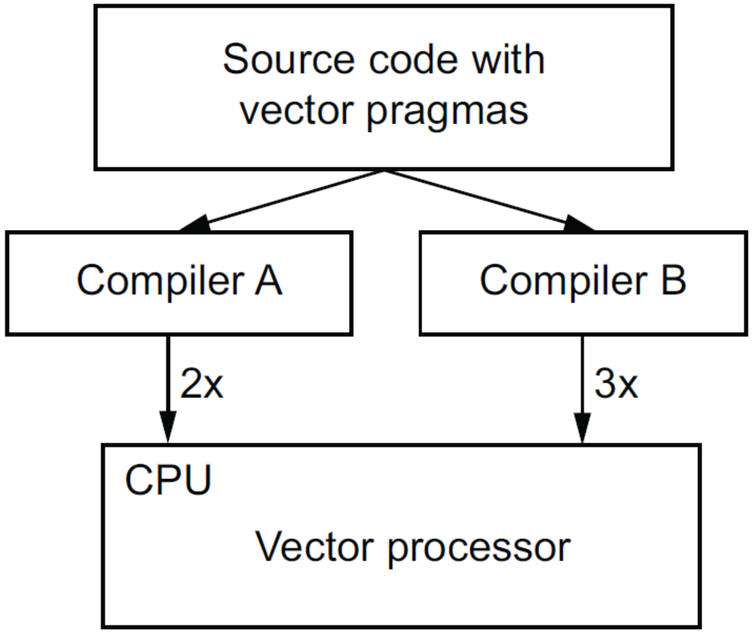


Figure 1.22 **Vector instructions in source code returning different performance levels from compilers**

Stream-based parallelization

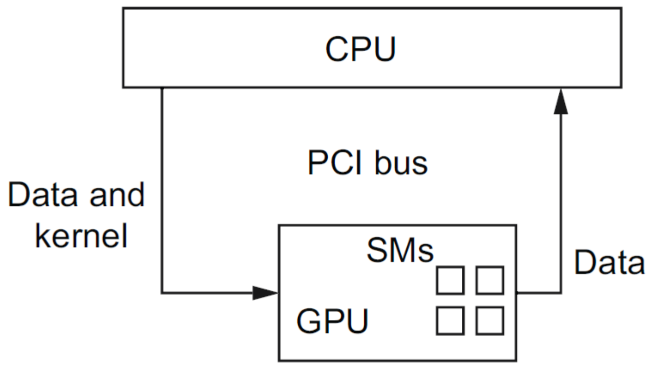


Figure 1.23 In the stream processing approach, data and compute kernel are offloaded to the GPU and its streaming multiprocessors. Processed data, or output, transfers back to the CPU for file IO or other work.

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;
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and 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];
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Shared Memory

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.
- Static threads
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.

Copyright © 2010, Elsevier Inc. All rights Reserved

Nondeterminism

```
. . .
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
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Nondeterminism

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

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` to `x`
5	Start other work	Store `x = 19`

Copyright © 2010, Elsevier Inc.
All rights Reserved

Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

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

Copyright © 2010, Elsevier Inc. All rights Reserved

busy-waiting

```
my_val = Compute_val ( my_rank ) ;
i f ( my_rank == 1)
    whi l e ( ! ok_for_1 ) ;  /* Busy−wait loop */
x += my_val ;  /* Critical section */
i f ( my_rank == 0)
    ok_for_1 = true ;  /* Let thread 1 update x */
```

Copyright © 2010, Elsevier Inc. All rights Reserved

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 ) ;
} elseif ( my_rank == 0) {
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
    printf ( "Process 0 > Received: %s\n" , message ) ;
}
```

Copyright © 2010, Elsevier Inc.
All rights Reserved

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 ] ;
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Input and Output

- In distributed memory programs, only process 0 will access *stdin*. In shared memory programs, only the master thread or thread 0 will access *stdin*.

- In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*.

Copyright © 2010, Elsevier Inc. All rights Reserved

Input and Output

- However, 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.

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Input and 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.

Copyright © 2010, Elsevier Inc.
All rights Reserved

Performance

Copyright © 2010, Elsevier Inc. All rights Reserved

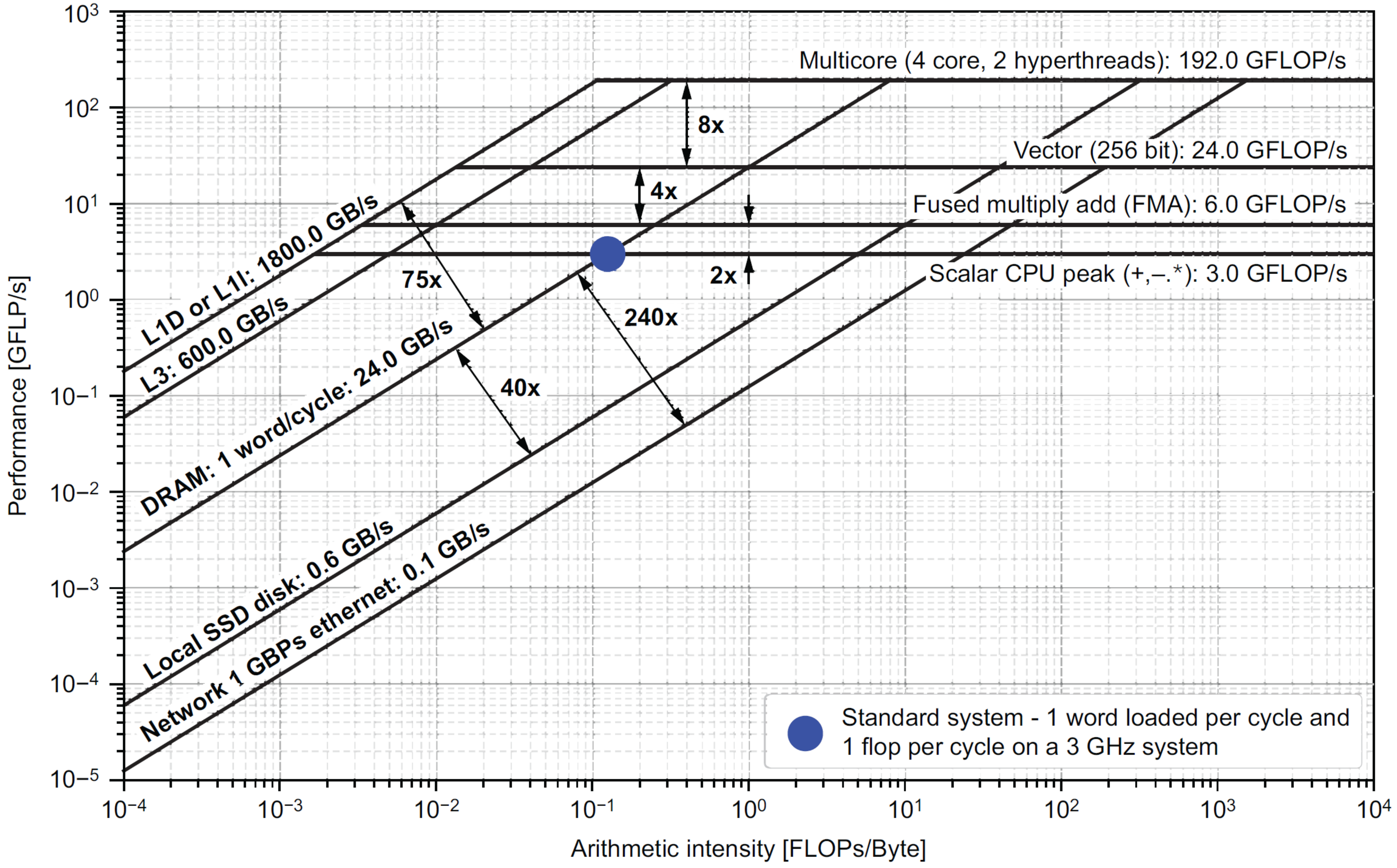
Performance Limitation

- Performance is limited by hardware:
  - Flops: floating point operations per seconds - CPU
  - Ops: operation per seconds – CPU
  - Memory bandwidth
  - Memory latency
  - Instruction queue (instruction cache)
  - Network
  - Disk

These limitations can be broken down into 2 categories:

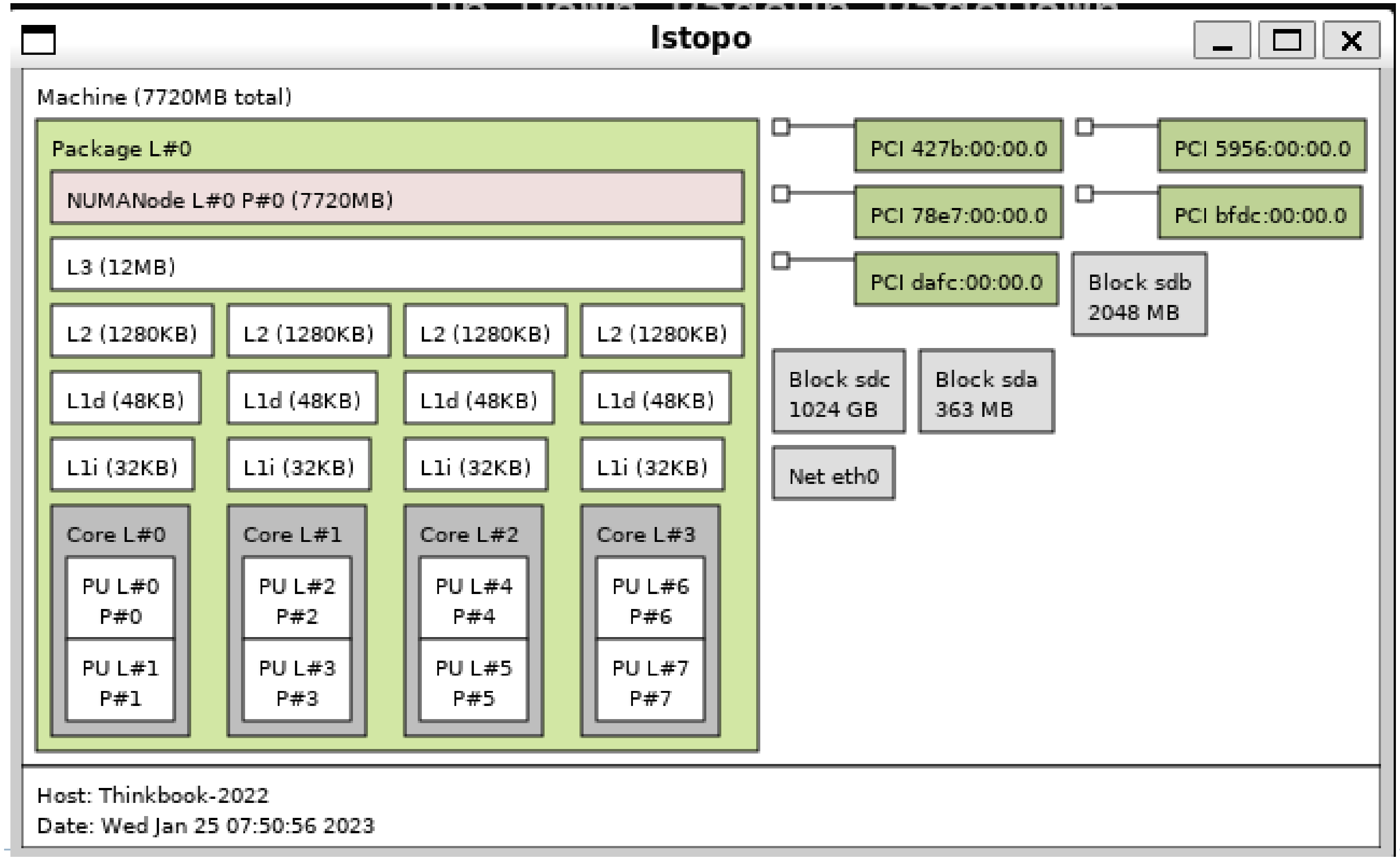
- Speed
- Feed/bandwidth

- Speed: how fast operations can be done.
  - To do the operations, you must get the data
- Feed: memory bandwidth, cache hierarchies, network & disk bandwidth
  - Applications that cannot get streaming behaviour, latency of memory, network and disk are more important.
  - Latency can be orders of magnitude slower than bandwidth
  - Organizing data so that it can be consumed in streaming pattern can yield dramatic speed up


Multicore (4 core, 2 hyperthreads): 192.0 GFLOP/s
Vector (256 bit): 24.0 GFLOP/s
Fused multiply add (FMA): 6.0 GFLOP/s
Scalar CPU peak (+,–.*): 3.0 GFLOP/s
8x
4x
2x
75x
240x
40x
L1D or L1I: 1800.0 GB/s
L3: 600.0 GB/s
DRAM: 1 word/cycle: 24.0 GB/s
Local SSD disk: 0.6 GB/s
Network 1 GBPs ethernet: 0.1 GB/s
Standard system - 1 word loaded per cycle and 1 flop per cycle on a 3 GHz system
Performance [GFLP/s]
Arithmetic intensity [FLOPs/Byte]
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}

Benchmarking

- Mengunakan lstopo: via hwloc package

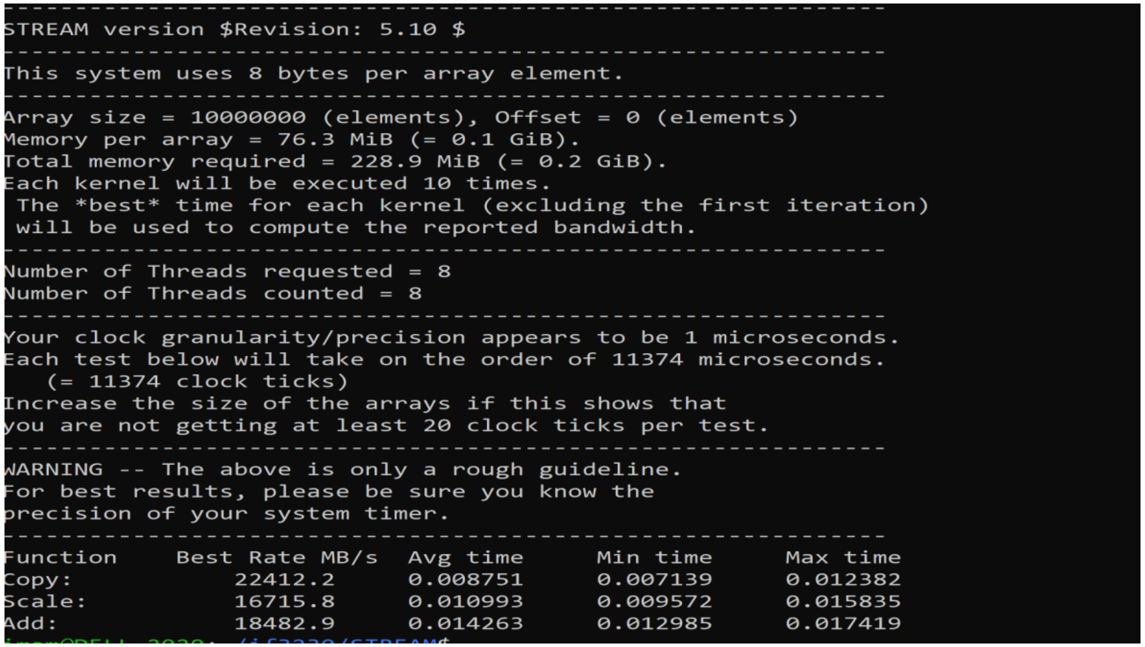


Host: Thinkbook-2022
Date: Wed Jan 25 07:50:56 2023

Benchmarking

- STREAM benchmark (https://github.com/jeffhammond/STREAM.git)
- Mengukur sustained memory bandwidth

		Bytes	Arithmetic Operations
Copy:	a(i) = b(i)	16	0
Scale:	a(i) = q*b(i)	16	1
Sum:	a(i) = b(i) + c(i)	24	1
Triad:	a(i) = b(i) + q*c(i)	24	2



```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 8
Number of Threads counted = 8
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 11374 microseconds.
   (= 11374 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           22412.2     0.008751     0.007139     0.012382
Scale:          16715.8     0.010993     0.009572     0.015835
Add:            18482.9     0.014263     0.012985     0.017419
```

Speedup

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

linear speedup

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Speedup of a parallel program

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

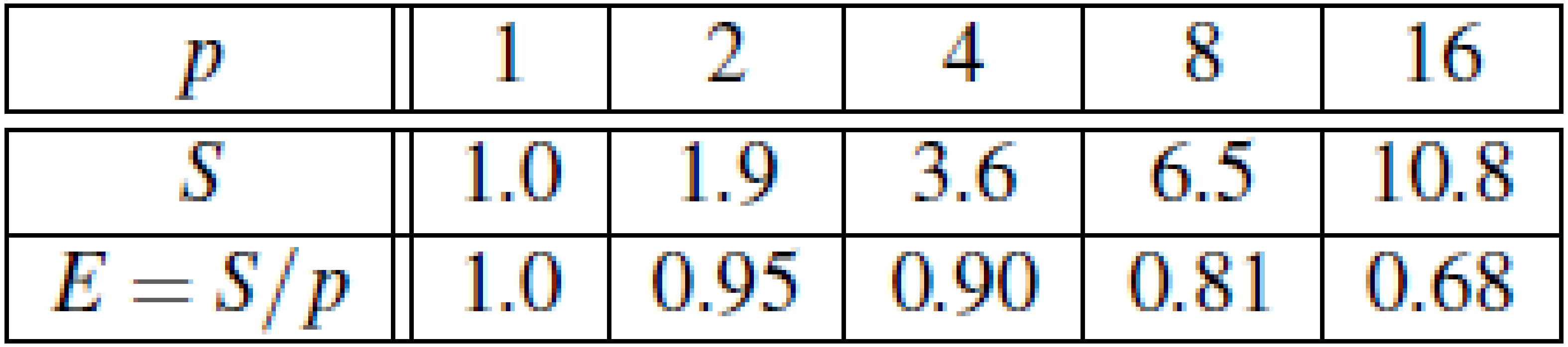
Copyright © 2010, Elsevier Inc.
All rights Reserved

Efficiency of a parallel program

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

Copyright © 2010, Elsevier Inc. All rights Reserved

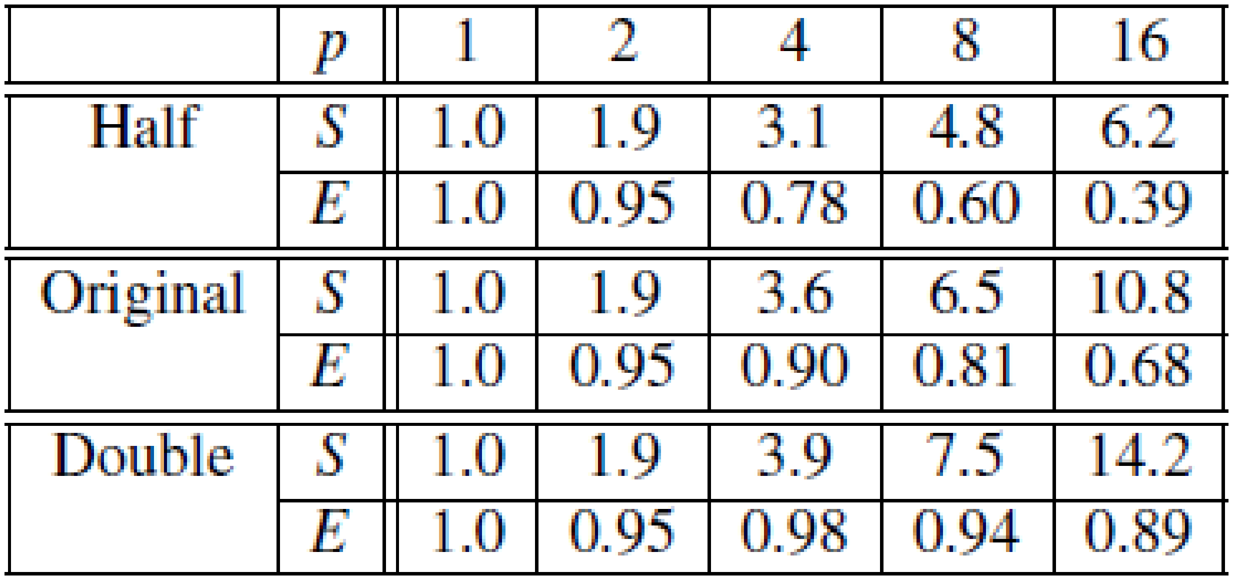
Speedups and efficiencies of a parallel program



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

Copyright © 2010, Elsevier Inc.
All rights Reserved

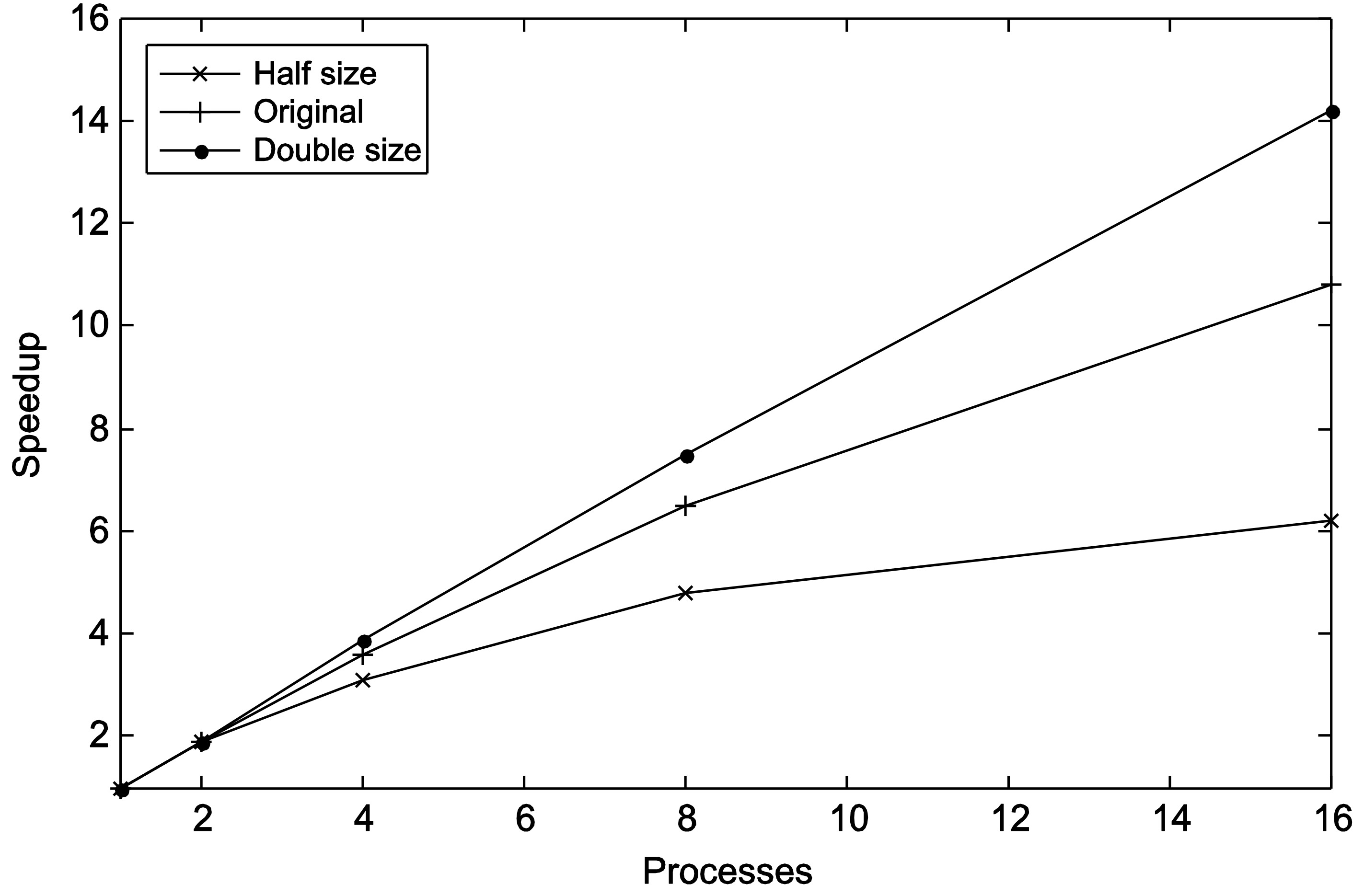
Speedups and efficiencies of parallel program on different problem sizes



	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

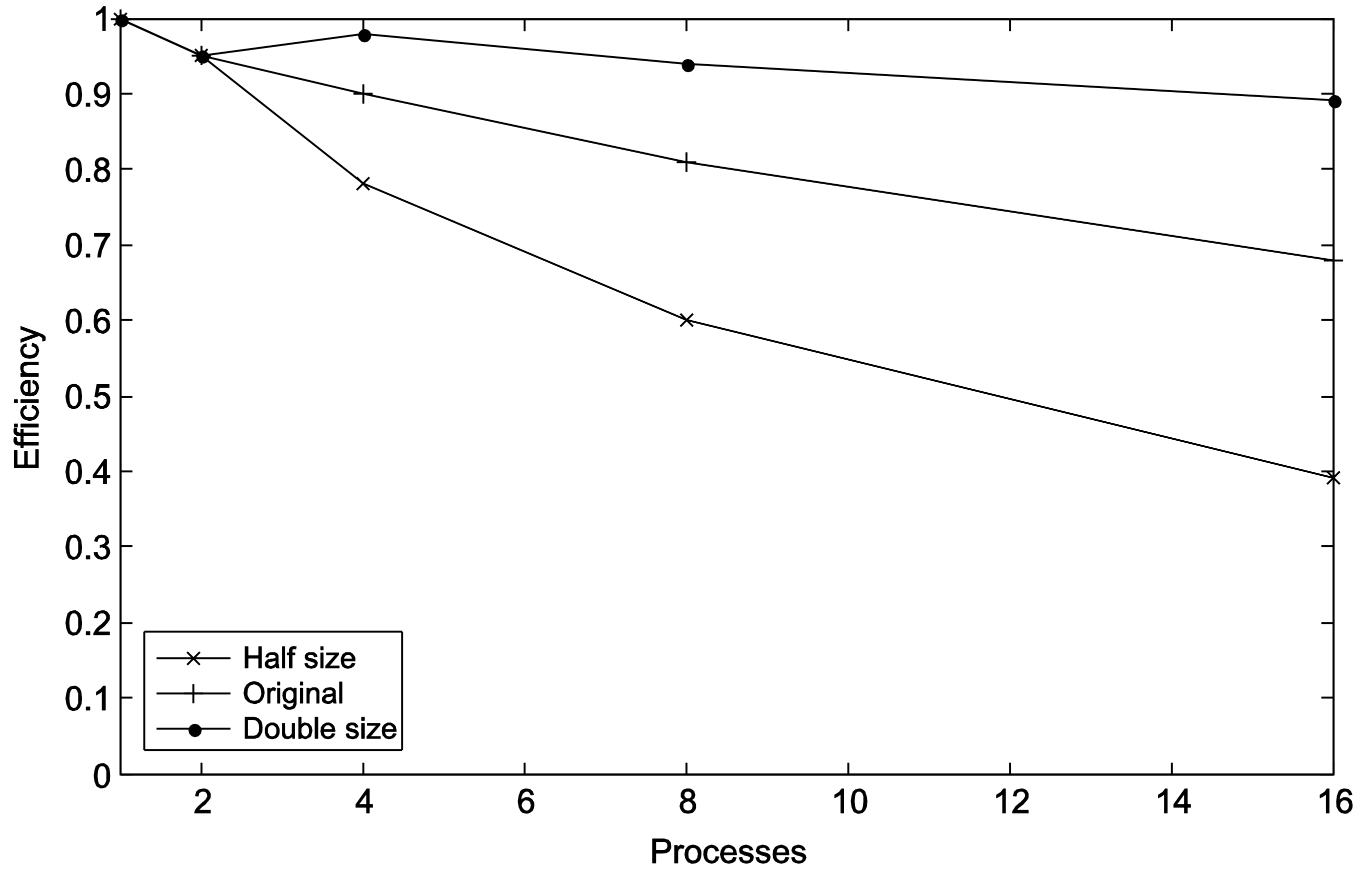
Copyright © 2010, Elsevier Inc. All rights Reserved

Speedup



Copyright © 2010, Elsevier Inc. All rights Reserved

Efficiency



Copyright © 2010, Elsevier Inc. All rights Reserved

Effect of overhead

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Amdahl's Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.
- S: Serial fraction, P: parallel fraction, S+P=1, N=jumlah processor

$$SpeedUp(N) = \frac{1}{S + \frac{P}{N}}$$

Copyright © 2010, Elsevier Inc. All rights Reserved

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

$$0.9 \times T_{serial} / p = 18 / p$$

Copyright © 2010, Elsevier Inc. All rights Reserved

Example (cont.)

- Runtime of "unparallelizable" part is

$$0.1 \times T_{serial} = 2$$

- Overall parallel run-time is

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Example (cont.)

- Speed up

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

Copyright © 2010, Elsevier Inc. All rights Reserved

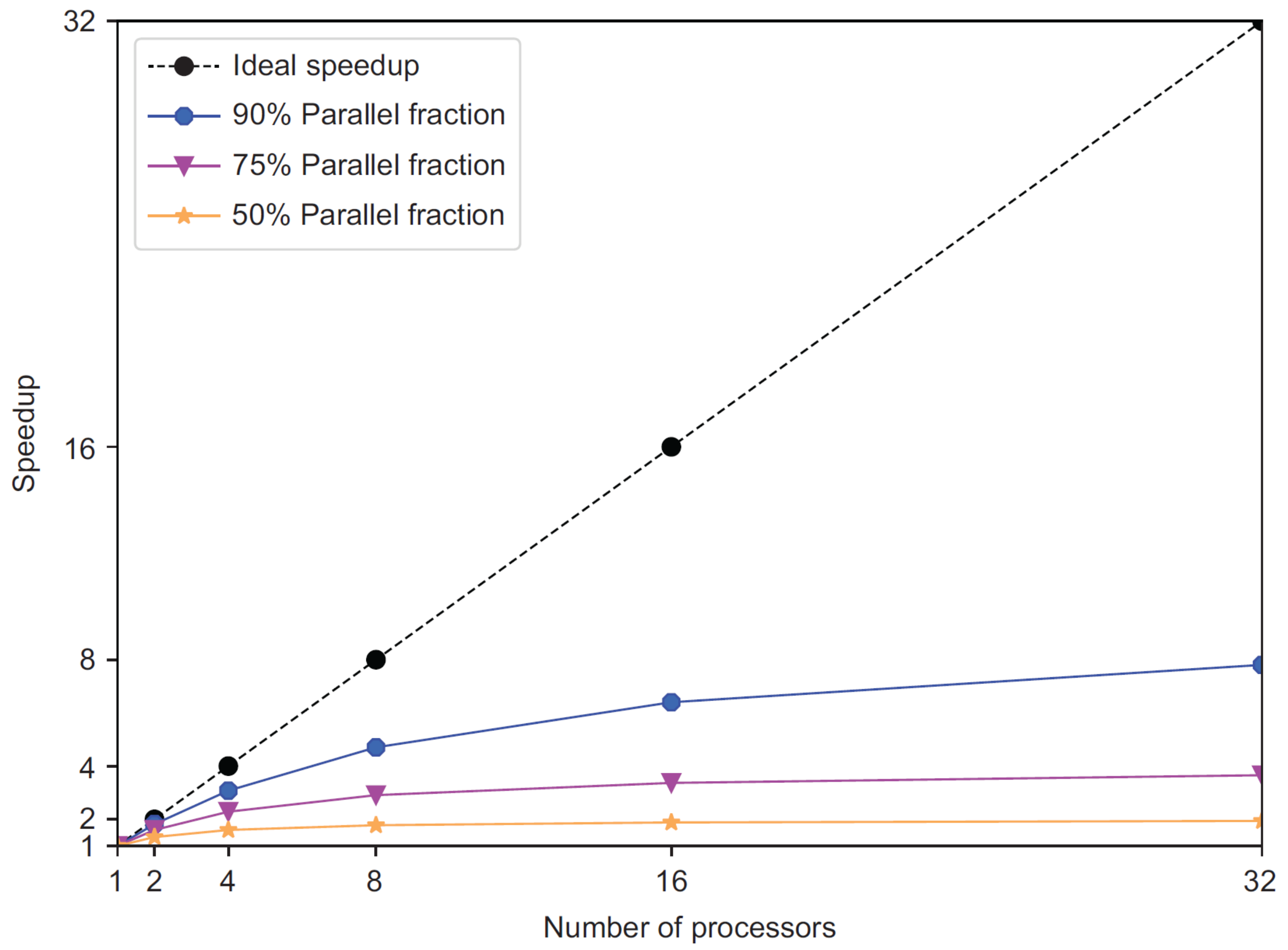


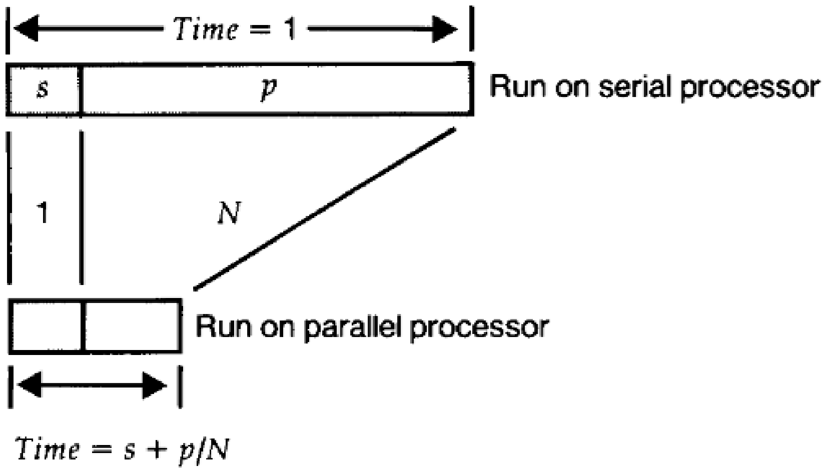
Figure 1.3 Speedup for a fixed-size problem according to Amdahl's Law is shown as a function of the number of processors. Lines show ideal speedup when 100% of an algorithm is parallelized, and for 90%, 75%, and 50%. Amdahl's Law states that speedup is limited by the fractions of code that remain serial.

Gustafson-Barsis’s Law

- Parallel code runs should increase the size of the problem as more processors are added
- If the size of the problems increase proportionally to the number of processors:

$$SpeedUp(N) = N - S * (N - 1)$$

Perhitungan speedup teoritis



Fixed-sized: Amdahl Law

FIGURE 2a. Fixed-Sized Model for $Speedup = 1/(s + p/N)$

Scaled-sized:
Gustafson-Barsis Law

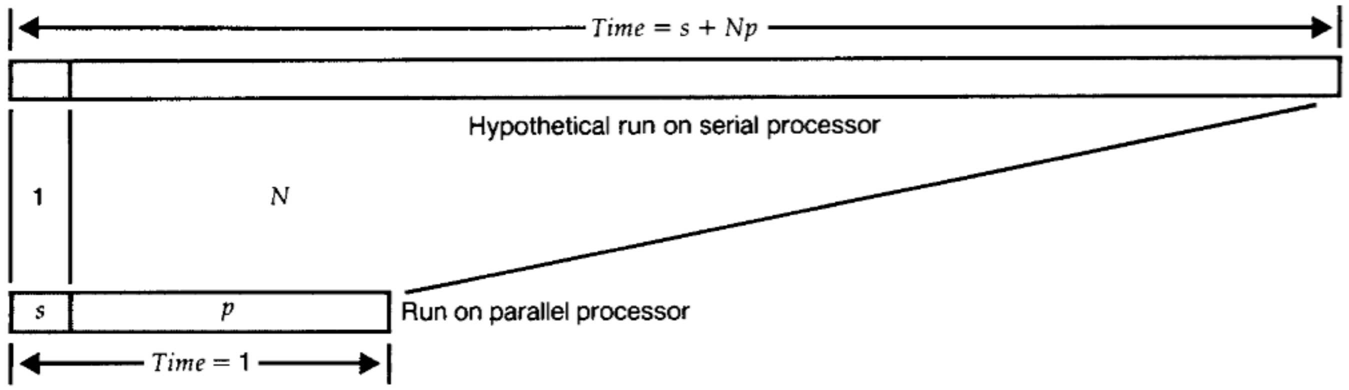
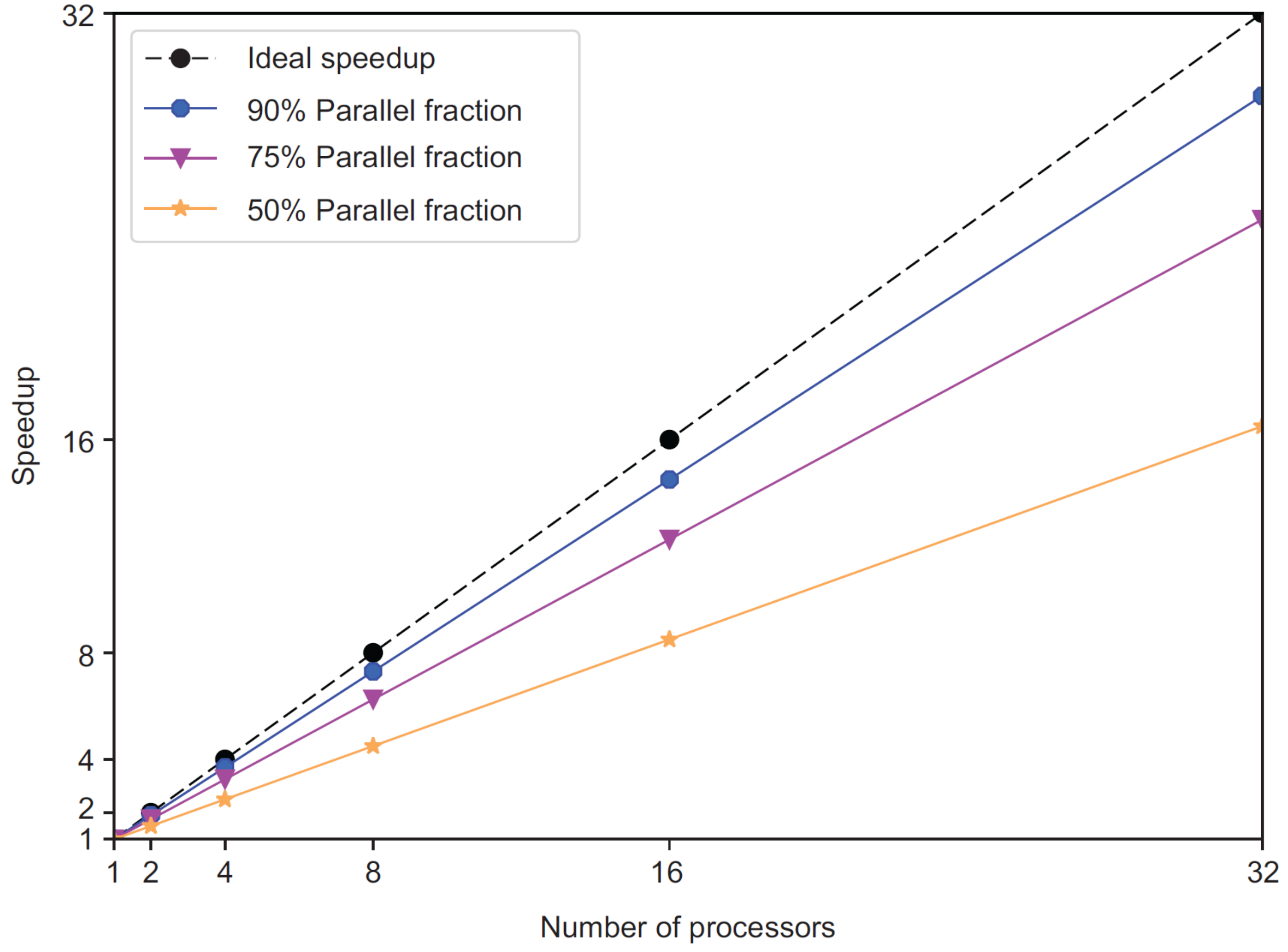

Time = s + Np
Hypothetical run on serial processor
1
N
s
p
Run on parallel processor
Time = 1

FIGURE 2b. Scaled-Sized Model for $Speedup = s + Np$

Speedup when the size of the problems grows



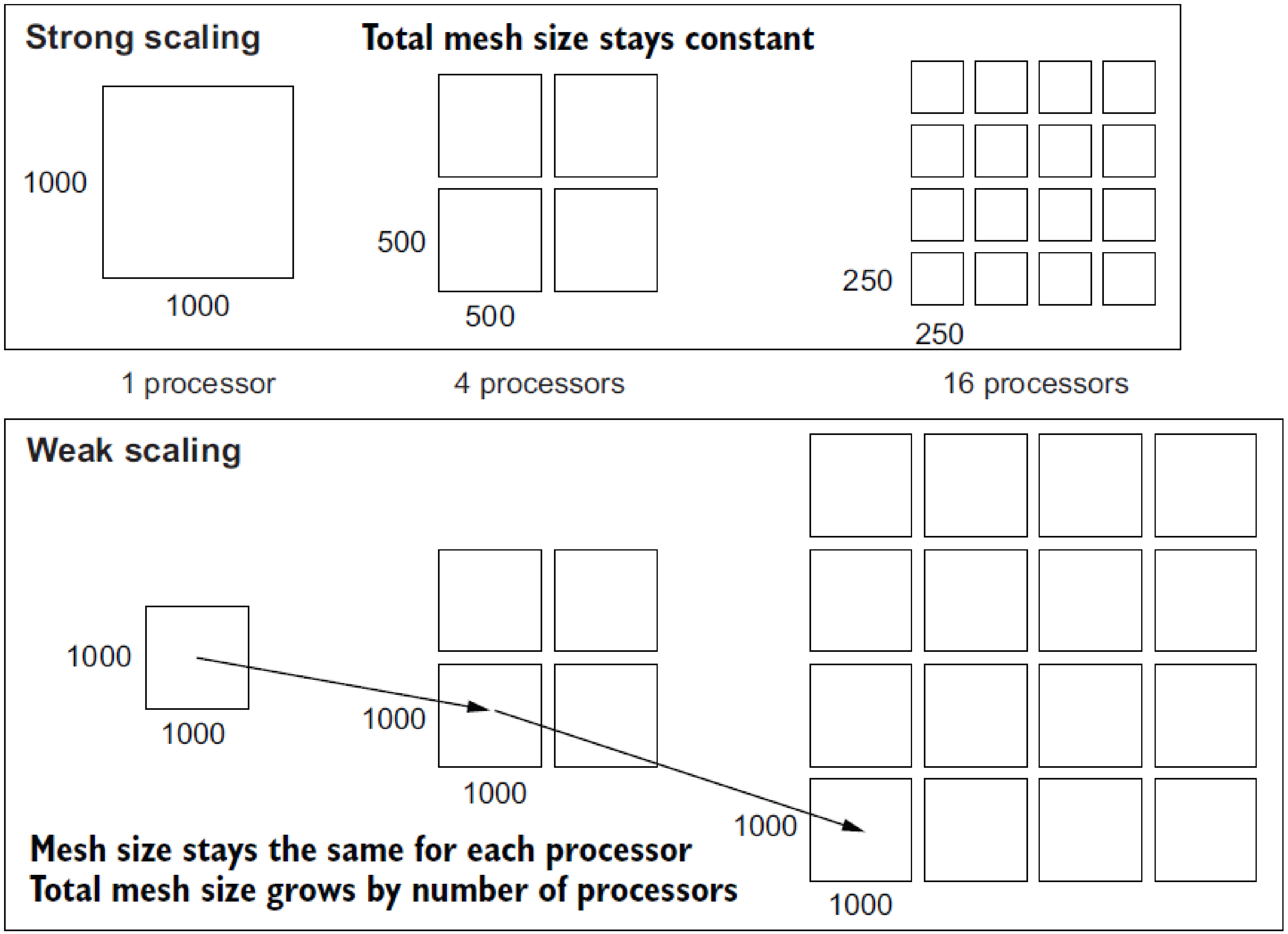
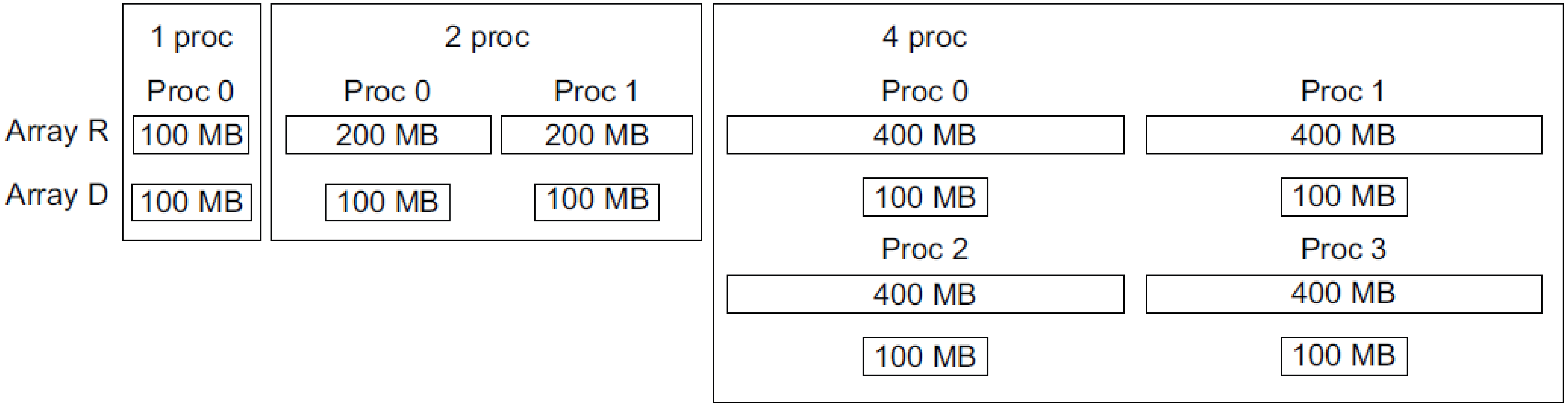

Strong scaling
Total mesh size stays constant
1000
1000
500
500
250
250
1 processor
4 processors
16 processors
Weak scaling
1000
1000
1000
1000
1000
1000
Mesh size stays the same for each processor
Total mesh size grows by number of processors

Figure 1.5 Strong scaling keeps the same overall size of a problem and splits it across additional processors. In weak scaling, the size of the mesh stays the same for each processor and the total size increases.

Memory sizes for weak scaling with replicated and distributed arrays



Array R – Array is replicated (copied) to every processor
Array D – Array is distributed across processors

Figure 1.6 Distributed arrays stay the same size as the problem and number of processors doubles (weak scaling). But replicated (copied) arrays need all the data on each processor, and memory grows rapidly with the number of processors. Even if the run time weakly scales (stays constant), the memory requirements limit scalability.

Scalability

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Taking Timings

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Taking Timings

```
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);
```

theoretical function

MPI_Wtime

omp_get_wtime

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

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);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

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.

Copyright © 2010, Elsevier Inc.
All rights Reserved

Concluding Remarks (2)

- 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.

Copyright © 2010, Elsevier Inc. All rights Reserved

Concluding Remarks (3)

- Performance
  - Speedup
  - Efficiency
  - Amdahl’s law
  - Scalability

Copyright © 2010, Elsevier Inc. All rights Reserved