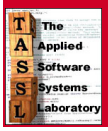
RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

ECE-451/ECE-566 - Introduction to Parallel and Distributed Programming

Lecture 2: Parallel architectures, Introduction to parallel software

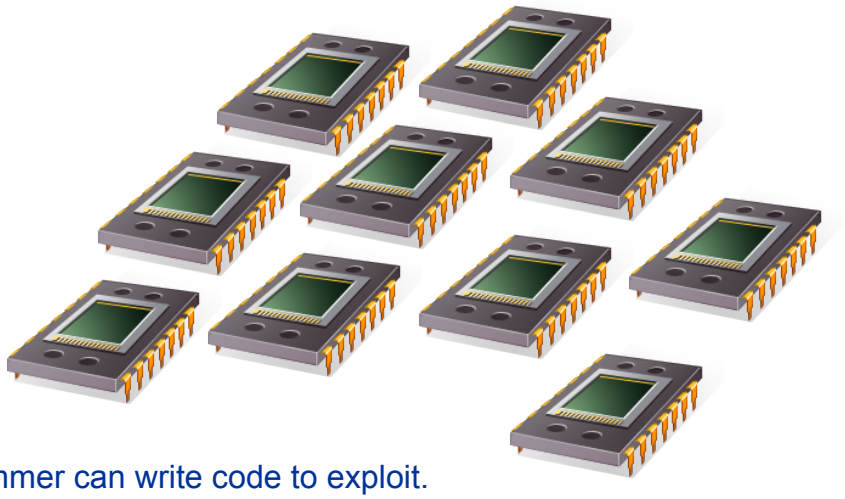
Department of Electrical & Computer Engineering
Rutgers University



CAC

Lecture Objectives/Organization

- Assignments information
- **Parallel hardware classification (Flynn's taxonomy)**
  - **Shared vs. distributed memory**
  - **Interconnection networks**
  - **Cache coherency**
- **Parallel software – SPMD**
  - **Nondeterminism (locks)**



A programmer can write code to exploit.

PARALLEL HARDWARE

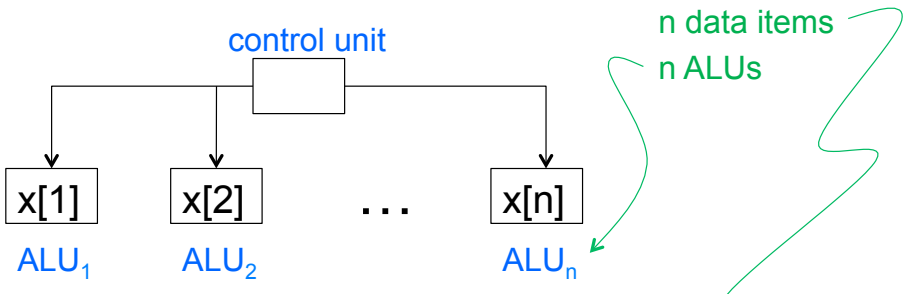
Flynn's Taxonomy

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

SIMD

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

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]	

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.

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.

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.

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.

Vector processors - Pros

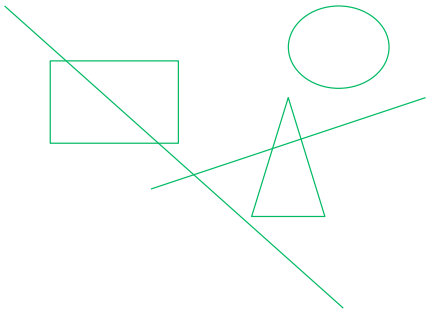
- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
- Compilers also can provide information about code that cannot be vectorized.
  - Helps the programmer re-evaluate code.
- High memory bandwidth.
- Uses every item in a cache line.

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)

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.



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.

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.

MIMD

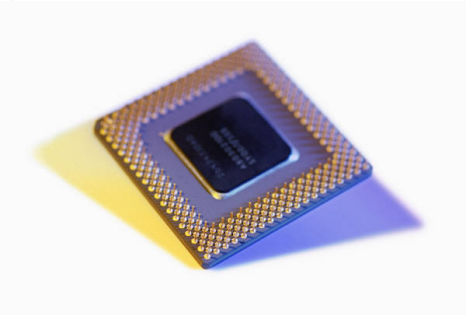
- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

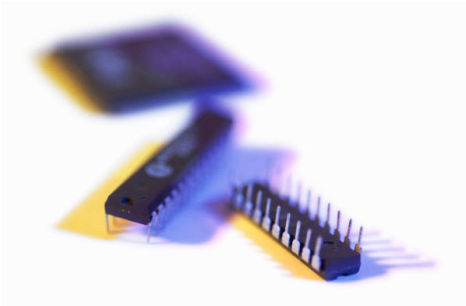
Shared Memory System (1)

- 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.

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)





Shared Memory System

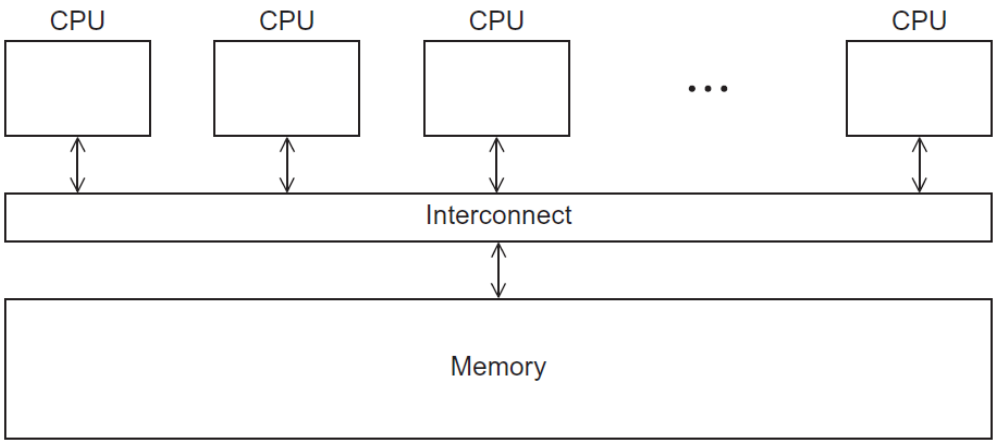


Figure 2.3

UMA multicore system

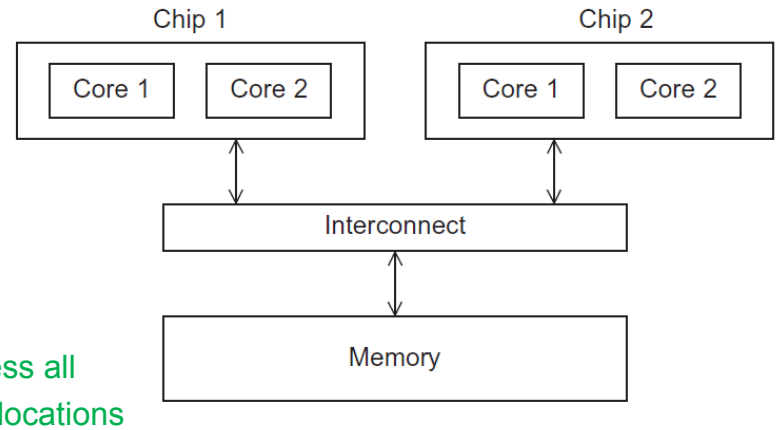


Figure 2.5

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

NUMA multicore system

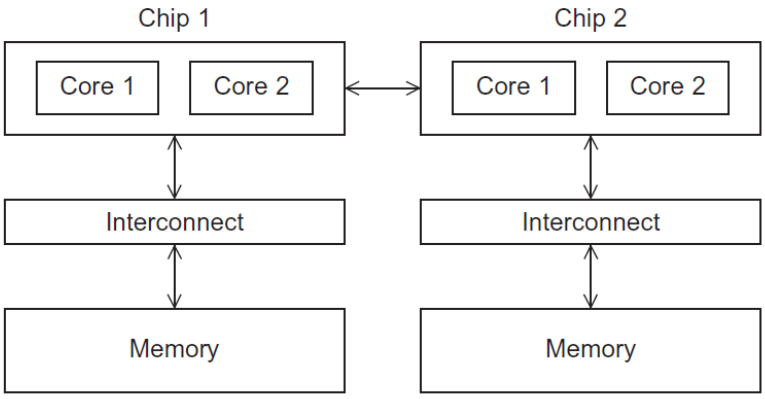


Figure 2.6

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

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

Distributed Memory System

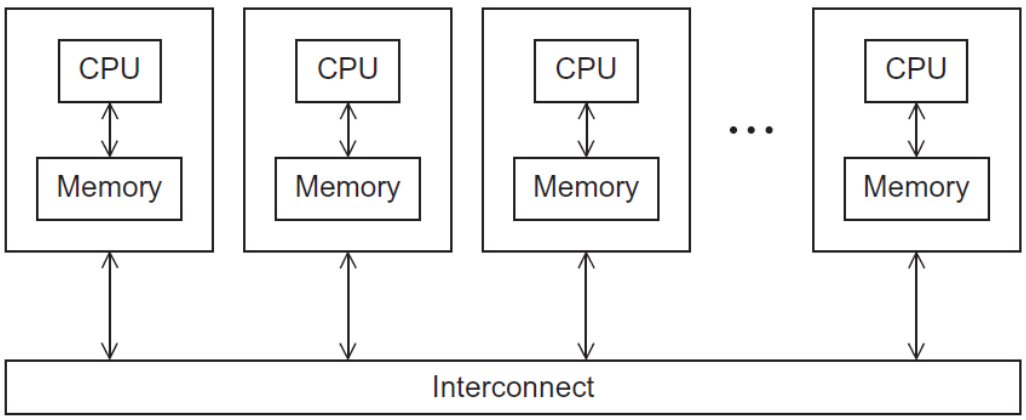


Figure 2.4

Interconnection networks

- Affects performance of both distributed and shared memory systems.
- Two categories:
  - Shared memory interconnects
  - Distributed memory interconnects

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.

Shared memory interconnects

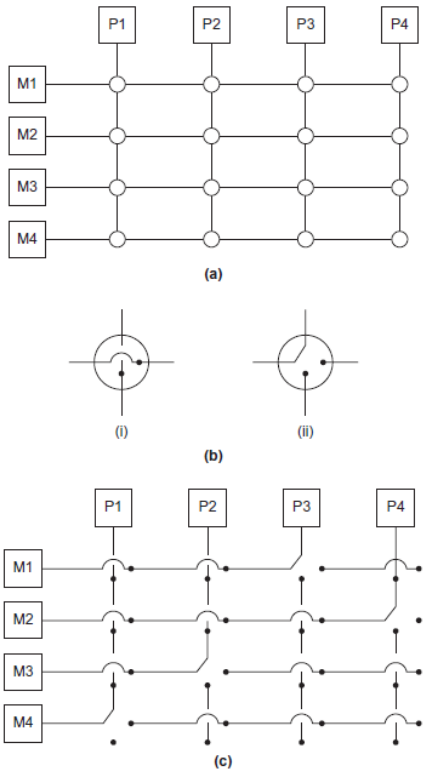
- Switched interconnect
  - Uses switches to control the routing of data among the connected devices.
  - Crossbar –
    - Allows simultaneous communication among different devices.
    - Faster than buses.
    - But the cost of the switches and links is relatively high.

Figure 2.7

(a)
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

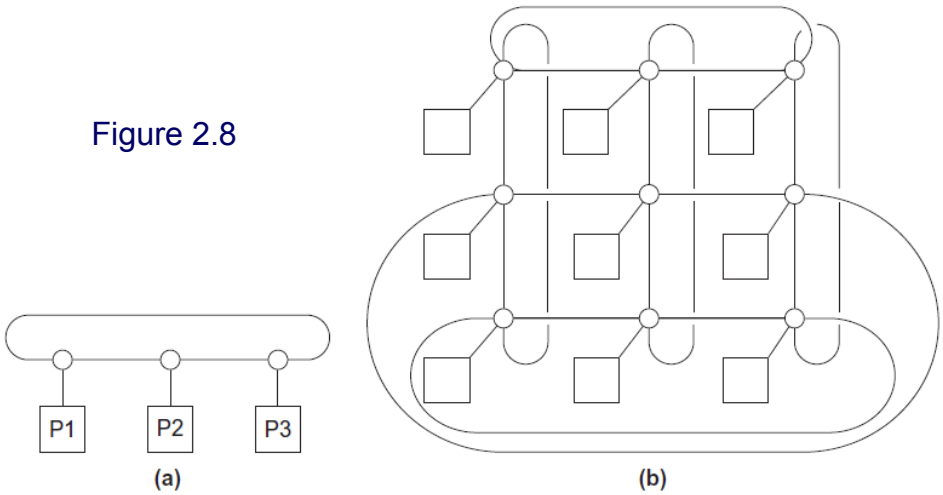


Distributed memory interconnects

- 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.

Direct interconnect

Figure 2.8



ring

toroidal mesh

Definitions

- Bandwidth
  - The rate at which a link can transmit data.
  - Usually given in megabits or megabytes per second.

Fully connected network

- Each switch is directly connected to every other switch.

impractical

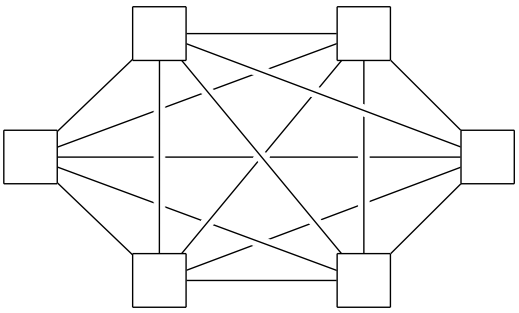


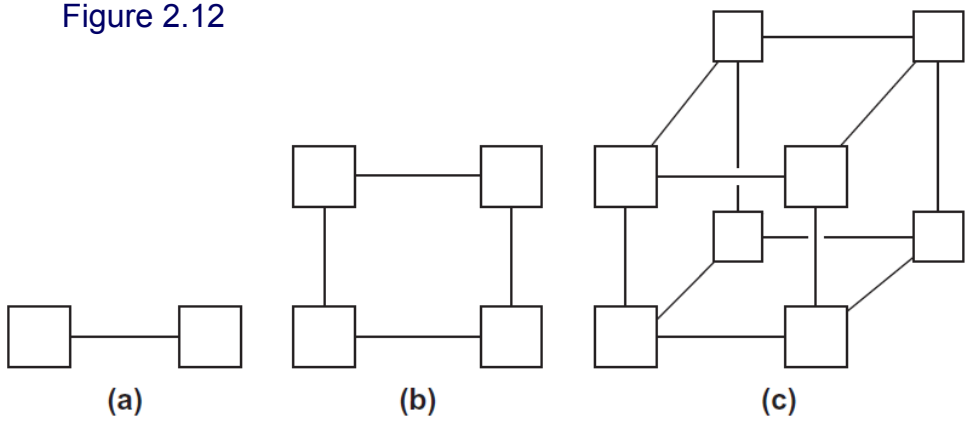
Figure 2.11

Hypercube

- Highly connected direct interconnect.
- 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.

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.

A generic indirect network

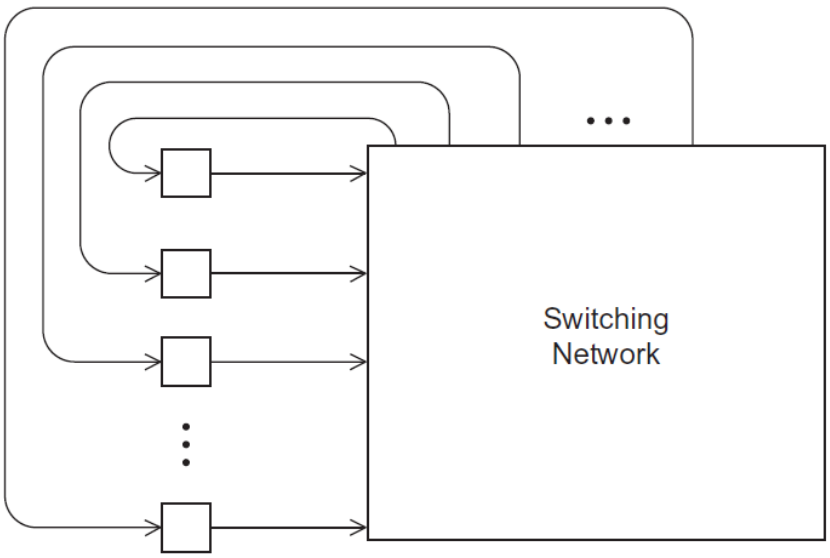


Figure 2.13

Crossbar interconnect for distributed memory

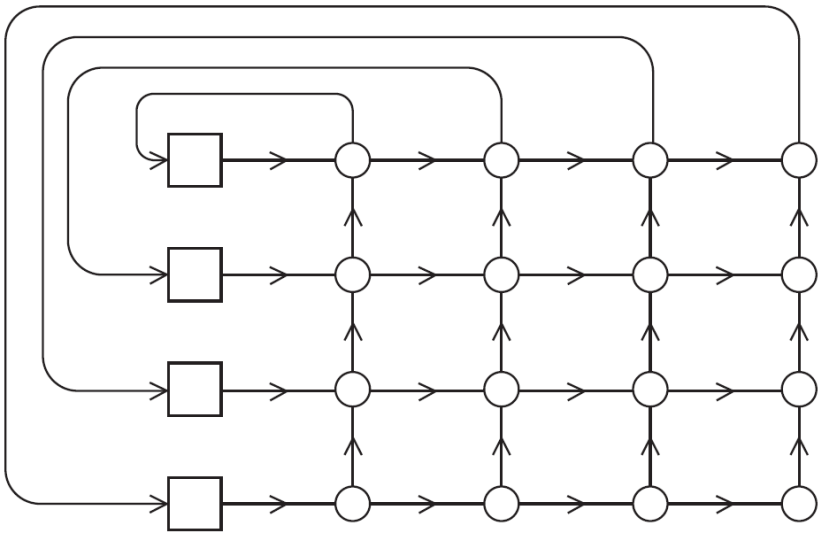


Figure 2.14

An omega network

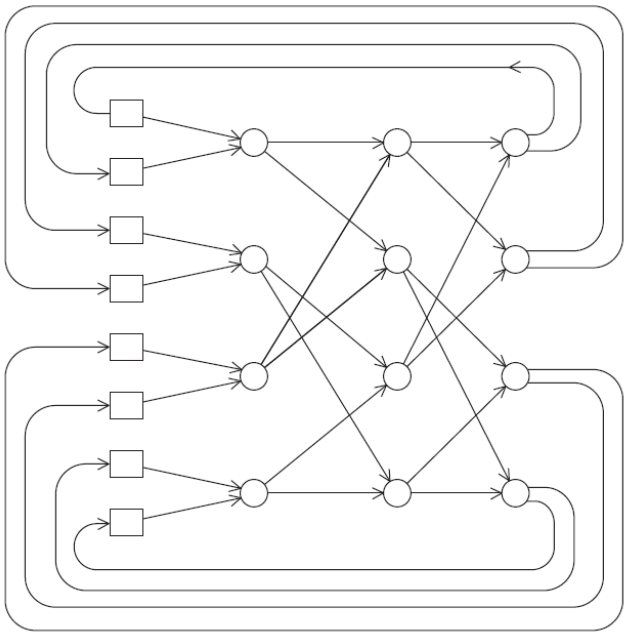


Figure 2.15

A switch in an omega network

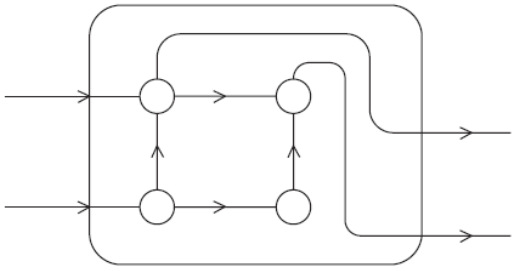
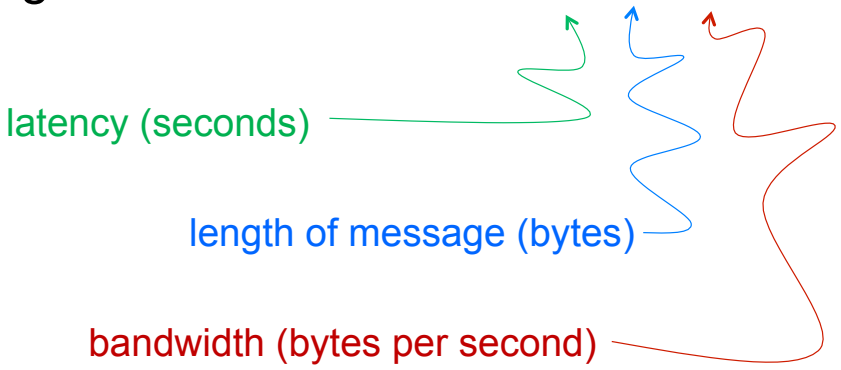


Figure 2.16

More definitions

- Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.
- Latency
  - 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.

Message transmission time = l + n / b



Cache coherence

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

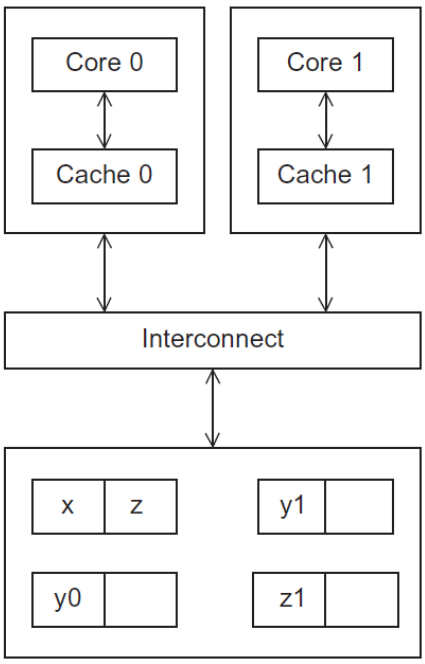


Figure 2.17

A shared memory system with two cores and two caches

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 = ???

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.

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.

PARALLEL SOFTWARE

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.

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;
```

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.

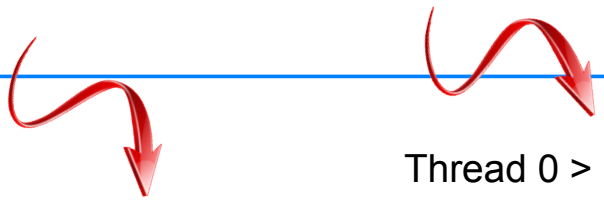
```
double x[n], y[n];
…
for (i = 0; i < n; i++)
    x[i] += y[i];
```


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.

Nondeterminism

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



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

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

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`

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 ) ;
```


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
*/
```

message-passing

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


Partitioned Global Address Space Languages

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

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*.

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.

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.

PERFORMANCE

Group Questions

- **When are vector processors useful?**
- **Similarity between vector processors and GPUs?**
- **How do you deal with cache coherency in shared mem?**
- **Limitations of buses?**
- **Transmission time in a network?**
- **In SPMD, how do you manage parallelism?**