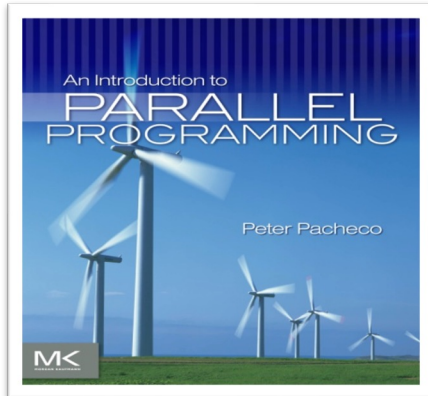




14013204-3 - PARALLEL COMPUTING



Chapter 2

Parallel Hardware



Roadmap

- 2.3 Parallel hardware
 - 2.3.1 Classifications of parallel computers
 - 2.3.2 SIMD systems
 - 2.3.3 MIMD systems
 - 2.3.4 Interconnection networks
 - 2.3.5 Cache coherence and False sharing

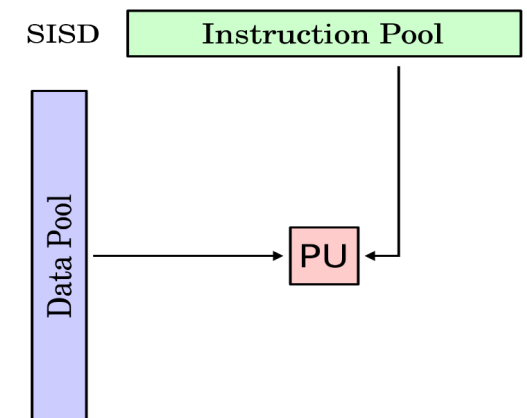
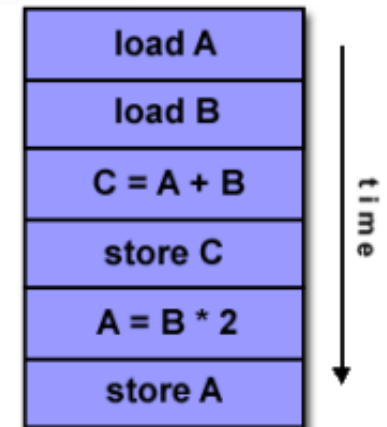
Flynn's Taxonomy

- Flynn classified computing architectures into four categories, based on the number of **instruction streams** and **data streams** they can simultaneously manage:

<i>classic von Neumann</i> (SISD) Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
(MISD) Multiple instruction stream Single data stream <i>not covered</i>	(MIMD) Multiple instruction stream Multiple data stream

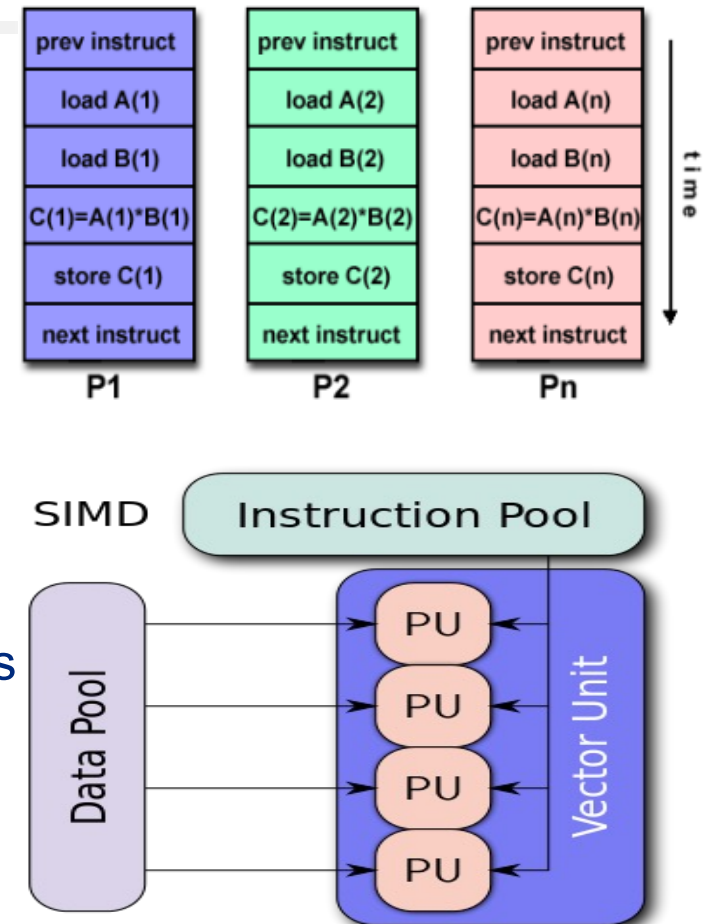
SISD

- It is a **sequential** computer (Not parallel) with a single processor → **classical von Neumann system**.
- **Single instruction**: Only one instruction stream is being acted on by the CPU during any **one clock cycle**.
- **Single data**: Only one data stream is being used as **input** during any one clock cycle.
- Deterministic execution.
- This is the oldest and until recently, the most **prevalent** form of computer.

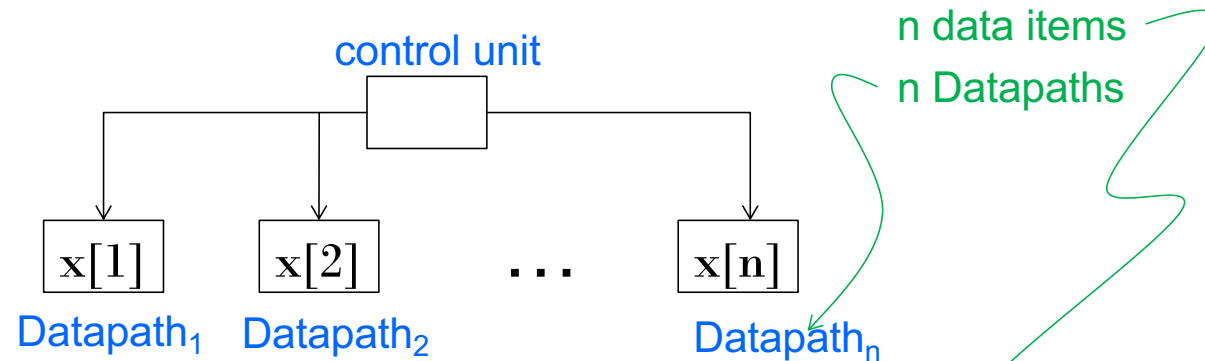


SIMD

- Parallelism is achieved by dividing data among the processors.
- Applies the same **single instruction** to **multiple data** items.
- An abstract SIMD system can be thought of as having a **single control unit** and **multiple datapaths**.
 - An instruction is broadcast **from the control unit** to **the datapaths**, and each datapath either applies the instruction to the current data item, or it is idle.
- Support **data parallelism**.



SIMD example



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

SIMD

- What if we don't have as many Datapaths as data items?
- Divide the work and process iteratively.
- Ex. $m = 4$ Datapaths and $n = 15$ data items.

Round	Datapath ₁	Datapath ₂	Datapath ₃	Datapath 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 Datapaths are required to execute the same instruction or remain idle → which can seriously **degrade the overall performance**.
 - For example, suppose we only want to carry out the addition if $y[i]$ is positive:
for ($i = 0; i < n ; i ++$)
 if ($y [i] > 0 . 0$) $x [i] += y [i] ;$
 - Some Datapaths will be idle if the condition is not true while others can proceed with the computation.
- In classic design, they must also operate **synchronously**, that is, each datapath **must wait** for the next instruction to be broadcast before proceeding.
- The Datapaths **have no instruction storage**, so a datapath can't delay the execution of an instruction by storing it for later execution.
- Efficient for large **data parallel** problems, but not other types of more complex parallel problems.

SIMD Systems

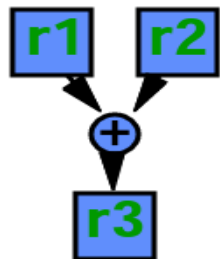
- SIMD systems have had a somewhat checkered history.
 - In the early 1990s, a maker of SIMD systems (Thinking Machines) was one of the largest manufacturers of parallel supercomputers.
 - However, by the late 1990s, the only widely produced SIMD systems were **vector processors**.
 - More recently, **graphics processing units (GPUs)**, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors (1)

- Operate on **arrays** or **vectors** of data while conventional CPUs operate on **individual data elements** or **scalars**.
 - The operand to the instructions are complete vectors instead of one element
- Typical recent systems have the following characteristics:
 1. **Vector registers:** Capable of storing a **vector of operands** and operating simultaneously on their contents.
 - The vector length is fixed by the system and can range from 4 to 256 64-bit elements.
 2. **Vectorized and pipelined functional units:** The **same operation** is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. → **vector operations are SIMD.**

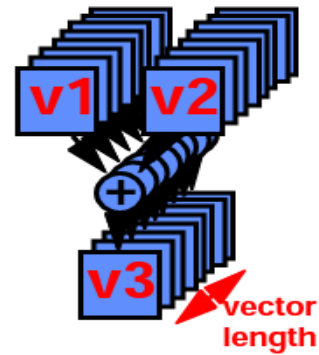
Vector processors (2)

SCALAR
(1 operation)



add r3, r1, r2

VECTOR
(N operations)



add.vv v3, v1, v2

Vector processors (3)

3. Vector instructions.

- Operate on vectors rather than scalars.
- If the vector length is `vector_length`, these instructions have the great virtue that a simple loop such as

```
for ( i = 0; i < n ; i ++)
```

```
  x [ i ] += y [ i ] ;
```

requires only a single load, add, and store for each block of `vector_length` elements, while a conventional system requires a load, add, and store for each element.

Vector processors (3)

4. optimized memory access

- 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
 - 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.

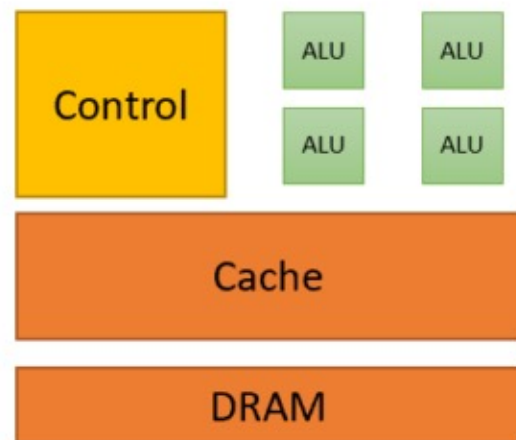
Vector processors - Cons

- 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**)

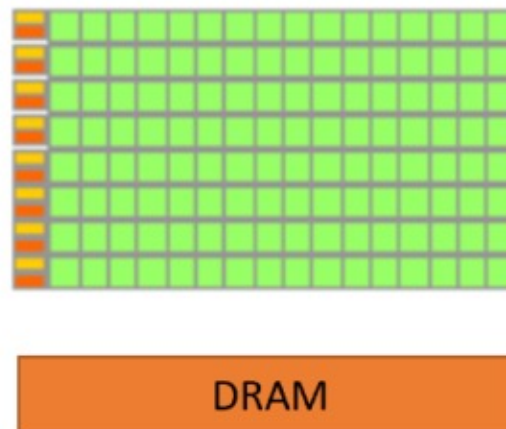


Graphics Processing Units (GPU)

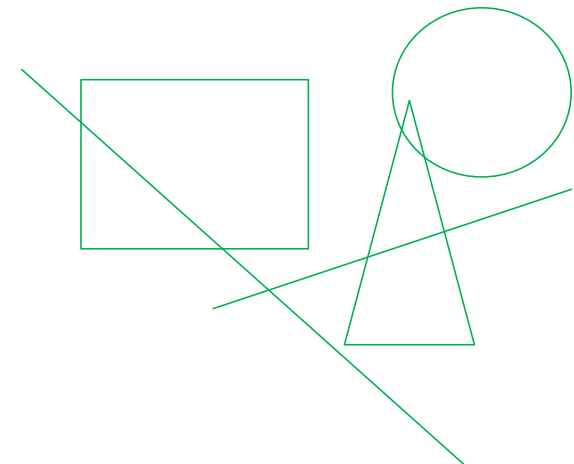
- Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.



CPU



GPU



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.

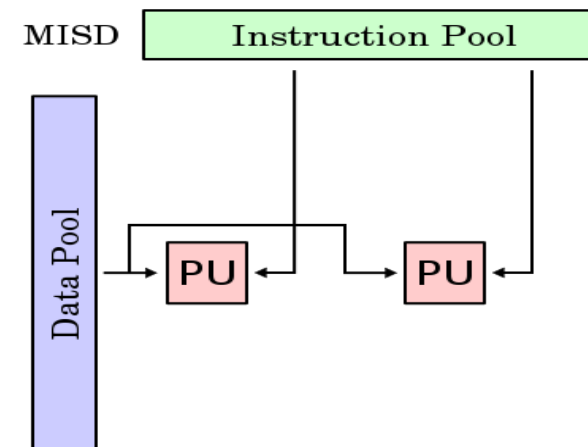
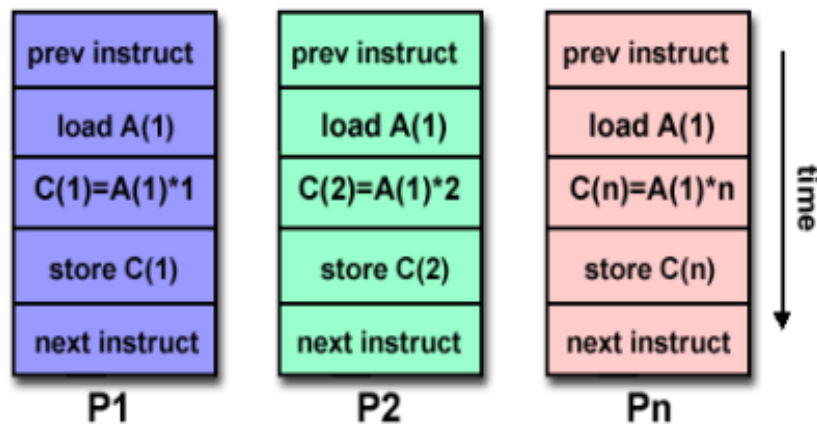


GPUs

- 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 GPUs use SIMD parallelism.
 - Although they are not pure SIMD systems.
- GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power.

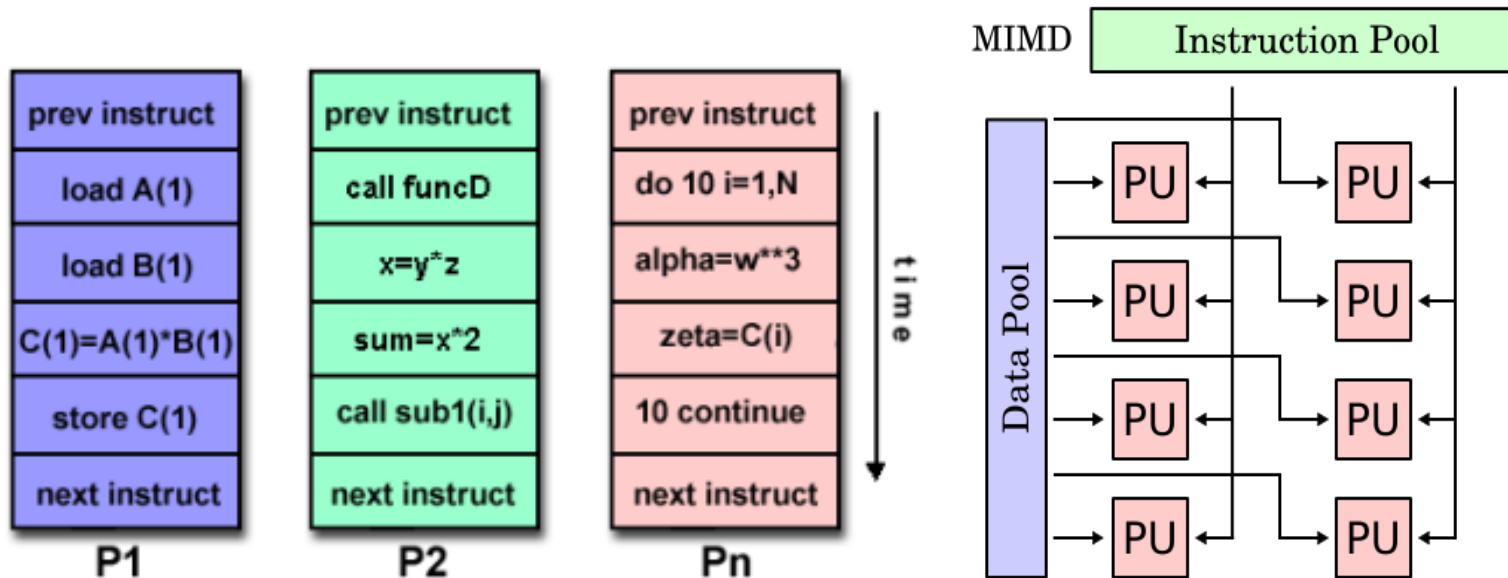
MISD

- It is not very common. Mainly mentioned for completeness.
- Multiple functional units or processors execute **multiple instructions** concurrently on the same **single data** stream.
 - A **single data** stream is **fed into multiple processing units**.
 - Each processing unit operates on the data independently via independent instruction streams.



MIMD

- Currently, the **most common type of parallel computer**. Most modern computers fall into this category.
- **Multiple Instruction**: Every processor may be executing a different instruction stream
- **Multiple Data**: Every processor may be working with a different data stream



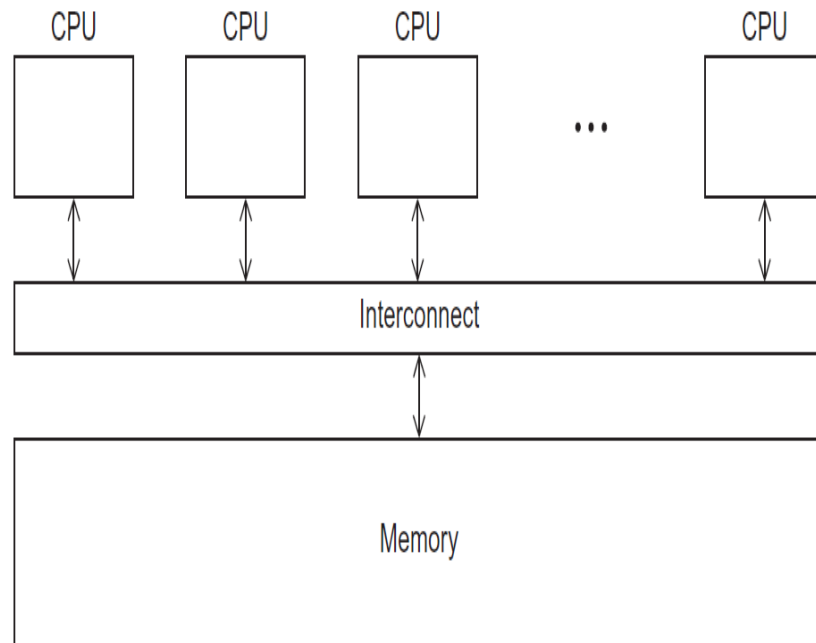
MIMD

- **MIMD systems** typically consist of a collection of **fully independent processing units** or **cores**, each of which has its own control unit and its own Datapath.
- Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace.
- Factors that promote the development of MIMD architectures:
 - Computers in this category offer support for a wider range of parallel algorithms
→ with wider applicability.
 - MIMD is extremely cost-effective
- We will focus our study on MIMD architectures ..

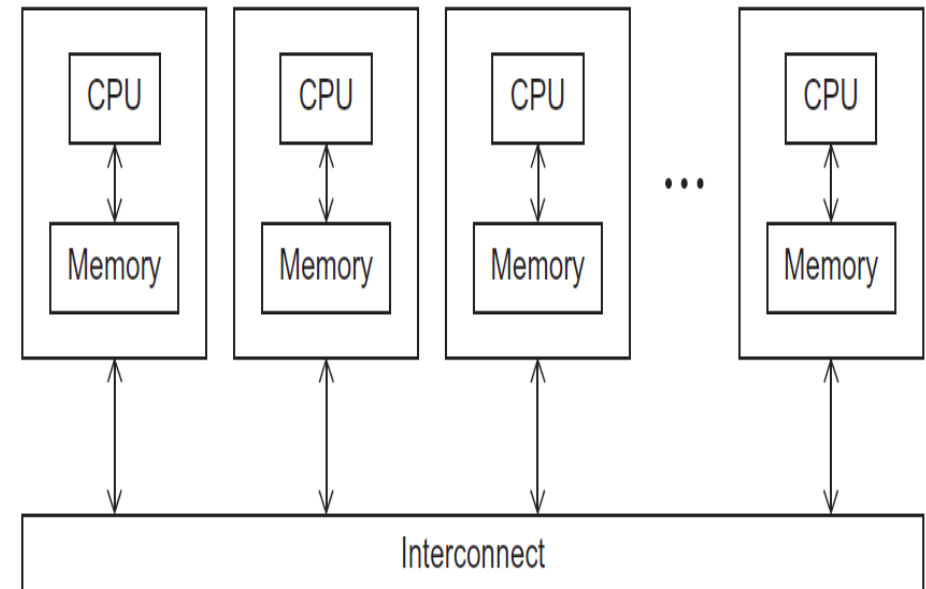
Classification of MIMD memory Architectures

- Two main categories of MIMD architectures, based on the organization of memory:
 1. **Shared memory architectures** → all cores **share access** to the **same** memory.
 2. **Distributed memory architectures** → each core has its **own** private memory, and the processor-memory pairs **communicate over an interconnection network**.
 - The processors usually communicate explicitly by **sending messages** or by using special functions that provide access to the memory of another processor.

Classification of MIMD memory Architectures



Shared memory architectures



Distributed memory architectures

Shared Memory System (1)

- All processors access all memory as **global address space**.
- Multiple processors can operate **independently**.
- **Changes** in a memory location caused by **one processor** are **visible** to all **other** processors.
- Terminology:
 - **Memory request**: The processor requires data from memory.
 - **Latency**: The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.(e.g., 200ms)
 - **Bandwidth**: Amount of data that can be transferred over a period of time (e.g., MB/sec).
 - Bandwidth determines how much data can be transferred at once, while latency measures the delay in data transmission.
 - **Contention**: Number of requests to memory per unit of time (requests/sec).

Shared Memory System (2)

- The most widely available shared-memory systems use one or more **multicore** processors.
- A multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have **private level 1 caches**, while other caches **may or may not** be shared between the cores.
- In such systems, the interconnect can either:
 - connect all the processors directly to the main memory → **UMA (uniform memory access)**.
 - or each processor can have a direct connection to a block of main memory, and the processors can access each other's blocks of main memory through special hardware built into the processors → **NUMA (nonuniform memory access)**.

Shared Memory System (3)

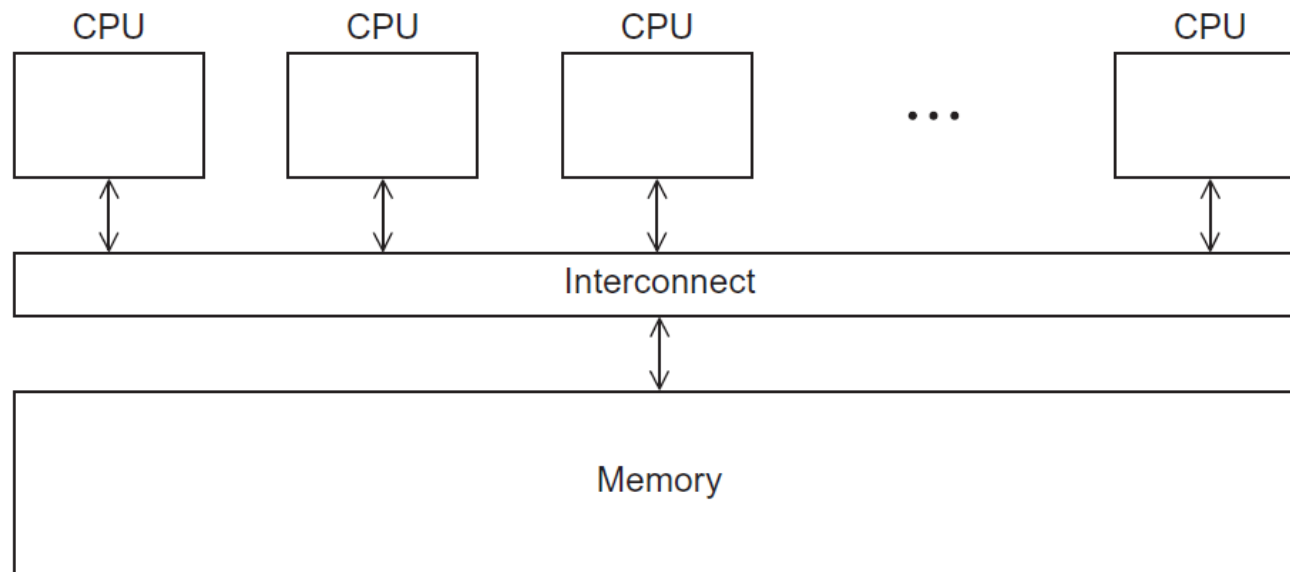


Figure 2.3

UMA multicore system

- Time to access all the memory locations will be the same for all the cores.
- Because access to shared memory is balanced, these systems are also called **SMP (symmetric multiprocessor)** systems.

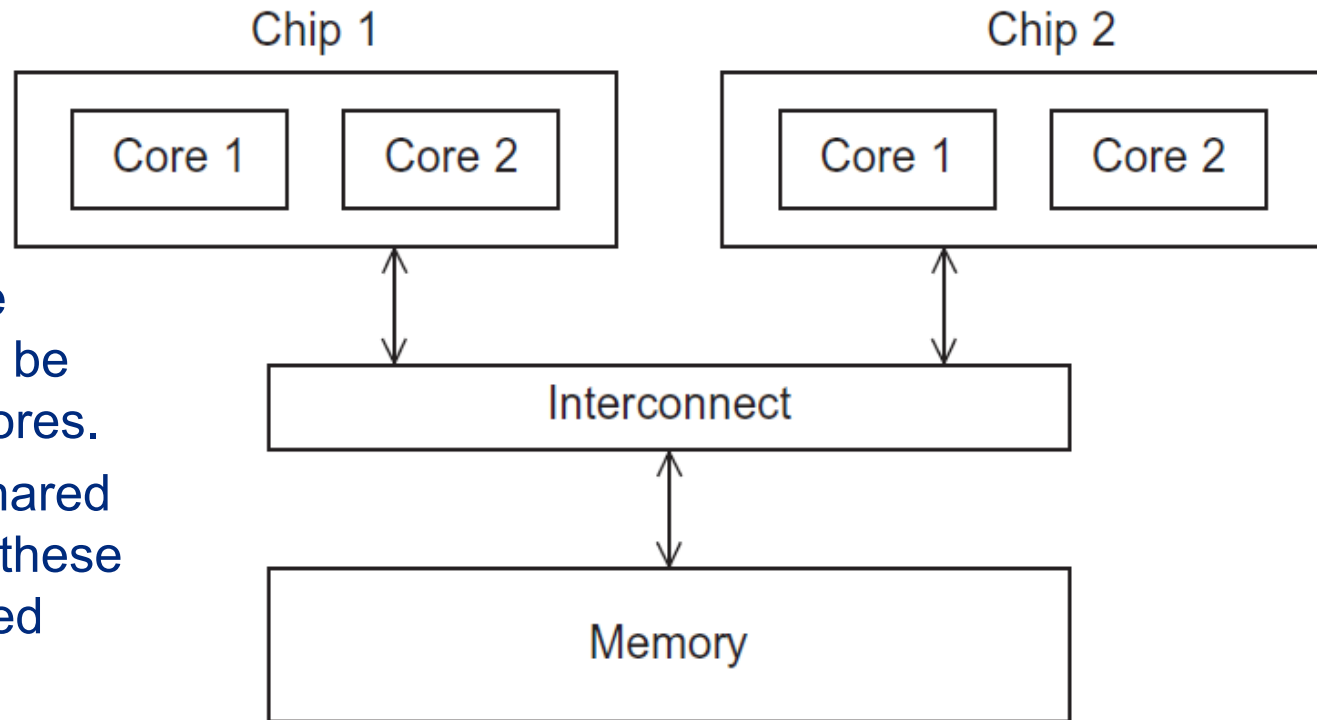


Figure 2.5

UMA multicore system

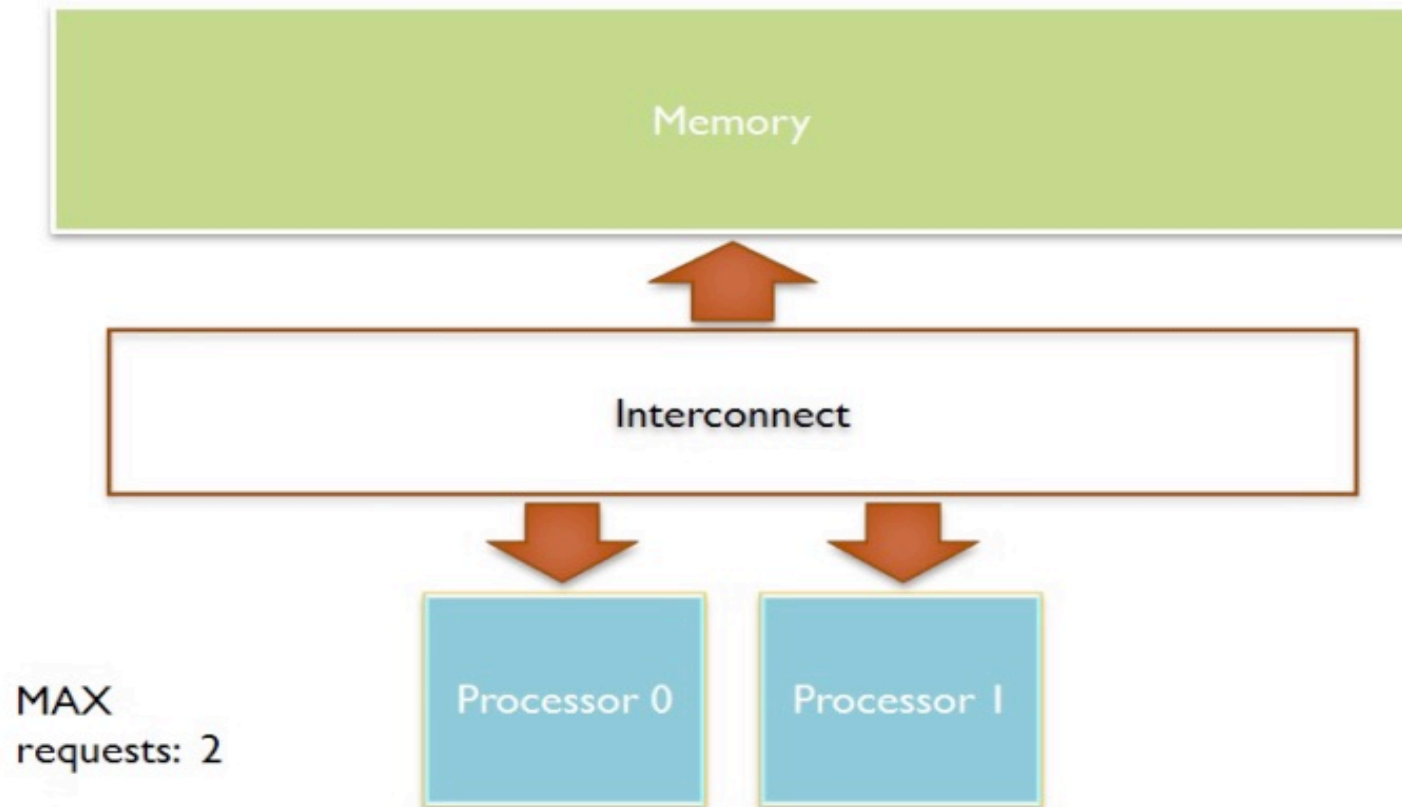
■ Advantage:

- **Low Latency:** Since the **Access time** to all memory locations is **equal** for all processors.
- **Easy to Implement:** because the programmer doesn't need to worry about different access times for different memory locations.
- **Low Cost**

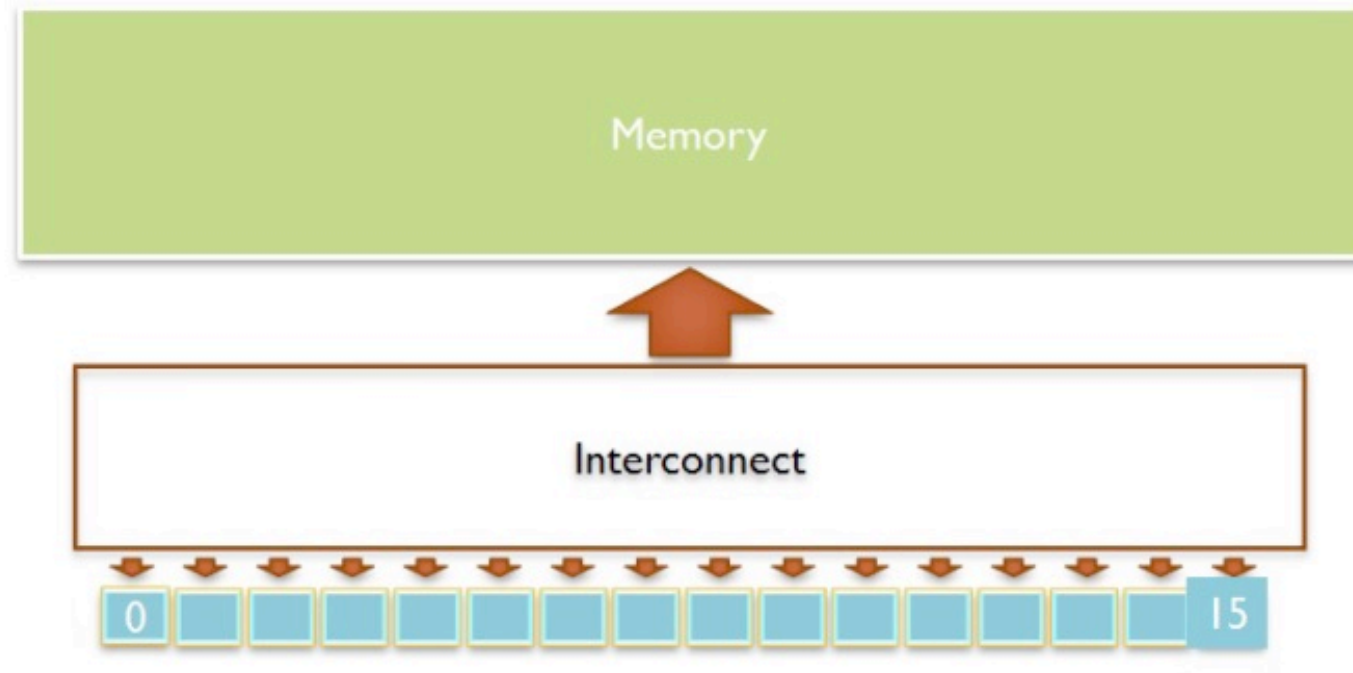
■ Disadvantage:

- **Limited Scalability:** All requests go to the central interconnect, adding more processors rises the **latency (delay)** due to increased **contention (competition for resources)** → **reduces performance** as the processors waste time waiting for memory requests.
- **Limited Bandwidth:** as all processors or cores share a single memory bus.
- **Limited Memory Capacity:** as all processors or cores share a single memory pool.

Example of Low Contention



Example of High Contention



MAX
requests: 16

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.

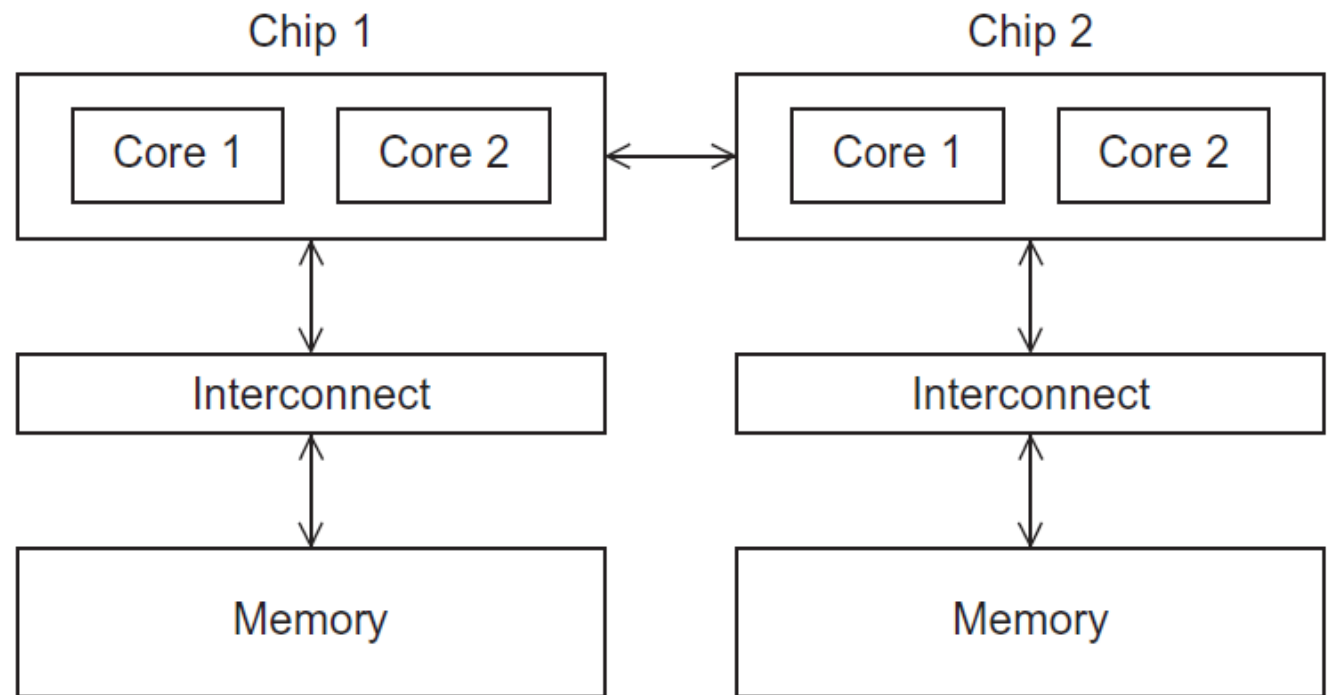
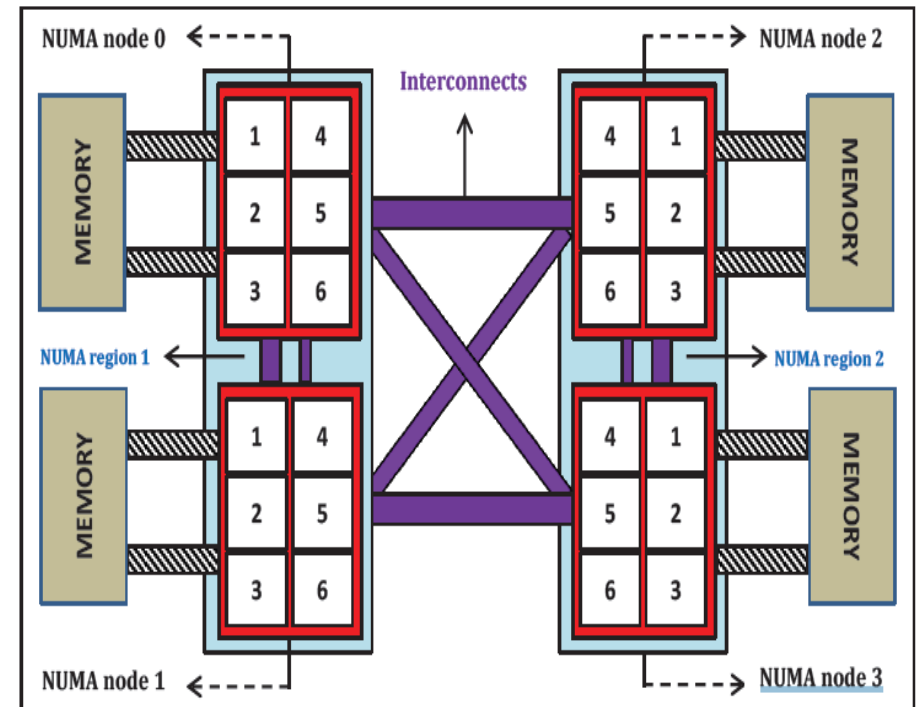


Figure 2.6

NUMA multicore system

- Terminology:
 - **NUMA node**: a group of cores that share the same memory.
 - **NUMA region**: a group of NUMA nodes.
 - **Remote Memory**: Memory request in which the core requires access to remote memory (e.g. on different nodes or regions) through interconnect.
 - **Local Memory**: Memory request in which the core requires access to local memory that is directly connected to the core (e.g. in the same NUMA node).



NUMA multicore system

- **Advantages:**

- **Improved performance:** by providing each processor with its own local memory, reduce access time, and better performance. This is called **data locality**.
- **Reduces contention** on interconnect, by allowing each processor to access its own local memory, reducing the need for multiple processors to access the same memory location.
- **Scalability:** NUMA systems are highly scalable and can handle large workloads by adding additional processors and memory nodes.

- **Disadvantages:**

- **Complexity:** complex to design and implement, as they require specialized hardware and software to manage memory access.
- **Higher cost:** can be more expensive.
- Requires more work from programmers to achieve high performance.
- **Performance variability:** In some cases, the performance of a NUMA system may be lower than that of a UMA system, especially if the workload requires frequent access to shared remote memory which can be slow to access.

Distributed Memory System(1)

- Distributed memory systems require a **communication network** to connect inter-processor memory.
- Processors have their **own local memory**. **Memory addresses in one processor do not map to another processor**, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently, and changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of **cache coherency** does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.

Distributed Memory System(2)

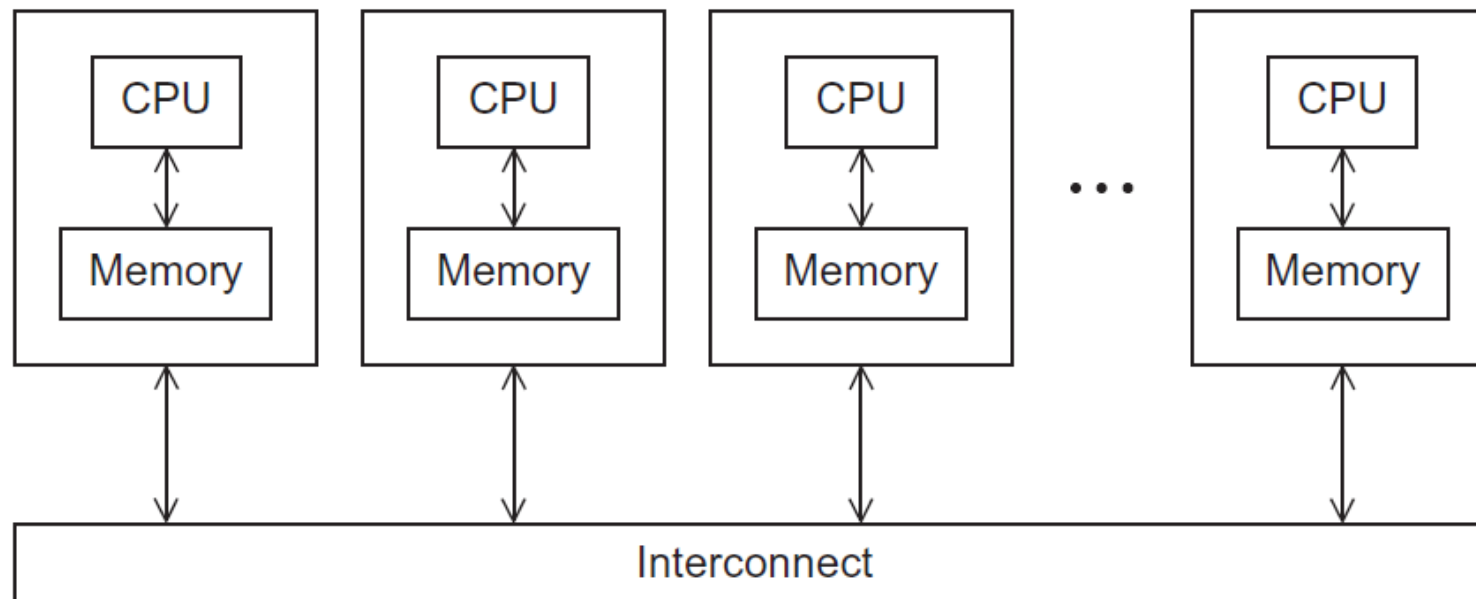


Figure 2.4

Hybrid Distributed-Shared Memory System (1)

■ Clusters (most popular)

- The most widely available distributed-memory systems.
- A collection of commodity systems, for example, PCs.
- Connected by a commodity interconnection network, for example, Ethernet
- **Nodes** of a cluster (the individual computational units joined by a communication network) are usually **shared-memory systems** with one or more multicore processors. *a.k.a. hybrid systems*
- Nowadays, it's usually understood that a cluster has **shared-memory nodes**

■ Grids

- Provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system is **heterogeneous**, that is, the individual nodes are built from different types of hardware.

Hybrid Distributed-Shared Memory System (2)

<u>Key</u>	<u>Cluster Computing</u>	<u>Grid Computing</u>
Computer Type	Nodes or computers have to be of the same type (hardware and operating systems). Cluster nodes are homogeneous .	Nodes or computers can be of different types(hardware and operating systems). Grid computers can be heterogeneous .
Task	Cluster computing is designed for high-performance computing tasks that require a dedicated and closely connected group of machines. It's suitable for applications that demand low-latency communication. (the whole system works as a single unit.)	Grid computing is employed for more diverse and loosely coupled tasks that can leverage resources from a wider network. It's ideal for large-scale projects that can be parallelized across multiple systems. (Each node in a Grid Computing network is independent and can be taken down or can be up at any time without impacting the functionality of other nodes.)
Location	Computers of Cluster computing are co-located and are connected by LAN (tightly coupled).	Computers of Grid Computing can be present at different locations (geographically dispersed) and connected by the Internet (loosely coupled).
Resource Manager	The cluster has a dedicated centralized resource manager, managing the resources of all the nodes.	Each node independently manages its own resources (decentralized resource management). It is designed to handle diverse workloads with varying resource requirements.

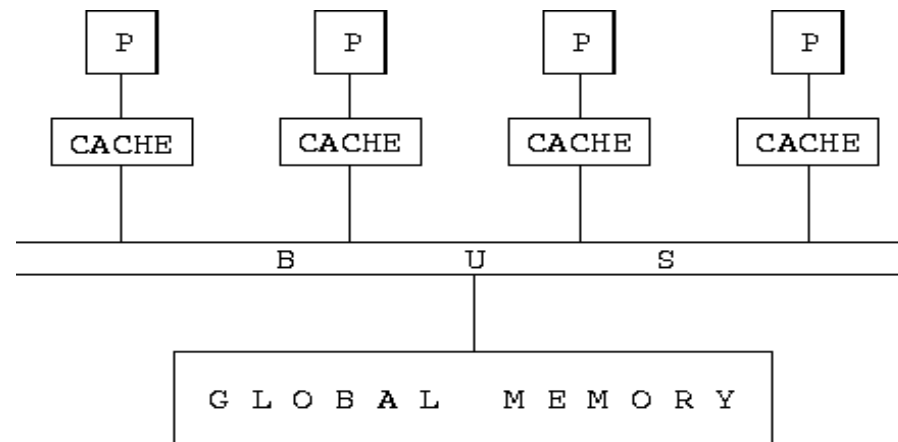
Interconnection networks

- Affects the performance of both distributed and shared memory systems.
 - Even if the processors and memory have virtually unlimited performance, a slow interconnect will **seriously degrade the overall performance** of all but the simplest parallel program.
- Two categories:
 1. Shared memory interconnects
 2. Distributed memory interconnects

Shared memory interconnects - Bus

■ Bus interconnect

- In the past, it was common for shared memory systems to use a bus to connect processors and memory.
- 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 the bus.



Shared memory interconnects - Bus

- All processing units and memory modules are connected to a single bus.
 - In each step, **at most one piece of data** can be written onto the bus.
 - This can be :
 - A request from a processing unit to read or write a memory value,
 - or it can be the response from the processing unit or memory module that holds the value.
- When a processing unit wants to read a memory word, it must first check to see if the bus is busy.
 - If the bus is **idle**, the processing unit puts the address of the desired word on the bus, issues the necessary control signals, and waits until the memory puts the desired word on the bus.
 - If, however, the bus is **busy** when a processing unit wants to read or write memory, the processing unit must wait until the bus becomes idle

Shared memory interconnects - Bus

■ Bus interconnect

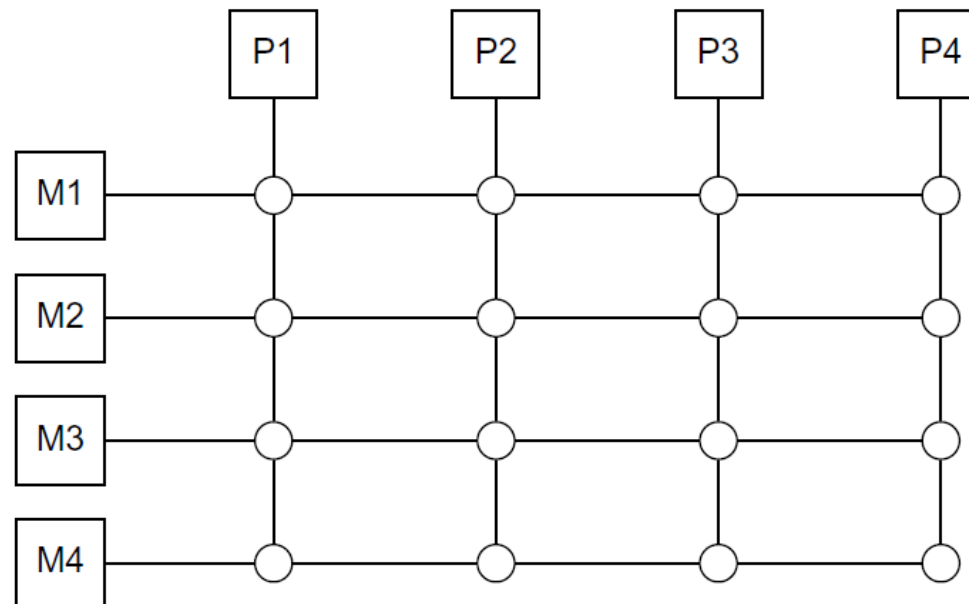
- (+) Buses have the virtue of **low cost** and **flexibility**; multiple devices can be connected to a bus with little additional cost.
- (-) As the number of devices connected to the bus increases, **contention for use of the bus increases, and performance decreases**. So, as the size of shared-memory systems has increased, **buses are being replaced by switched** interconnects.

Shared memory interconnects - Switched

- Switched interconnect
 - Uses switches to control the **routing of data** among the connected devices.
 - **Crossbar** –
 - A relatively simple and powerful switched interconnect.
 - Allows simultaneous communication among different devices.
 - (+) Faster than buses.
 - (-) But the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Shared memory interconnects - Switched

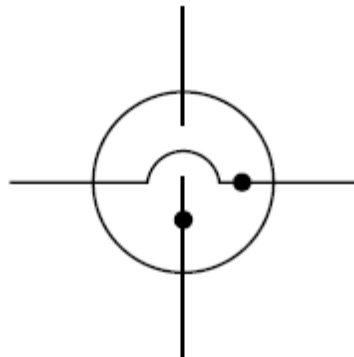
- The lines are **bidirectional communication links**, the squares are **cores or memory modules**, and the **circles** are **switches**.



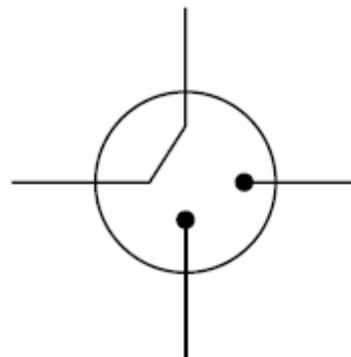
A crossbar switch connecting four processors (P_i) and four memory modules (M_j)

Shared memory interconnects - Switched

- The individual switches can assume one of the **two configurations** shown in Figure below.
- There will only be a **conflict** between **two cores attempting to access memory** if the two cores attempt to simultaneously access the **same memory module**.



(i)

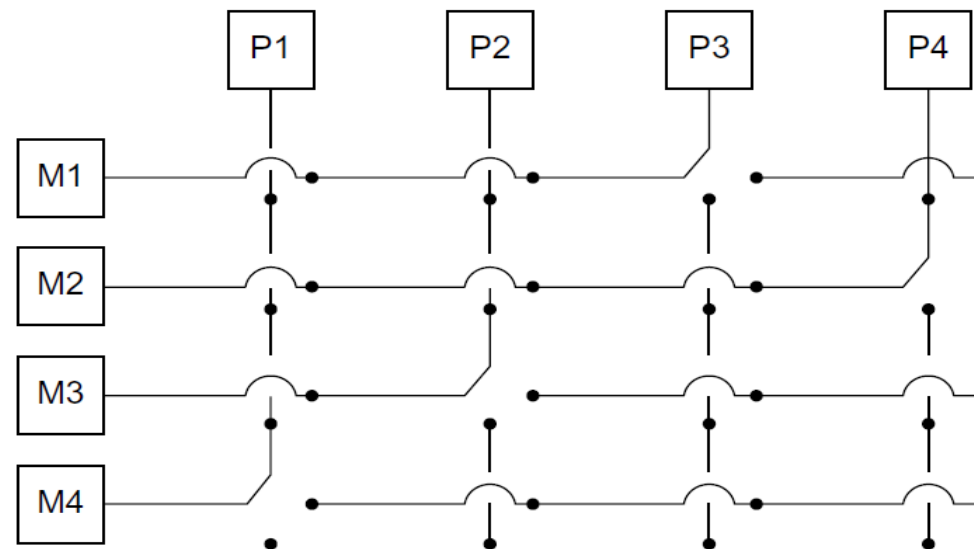


(ii)

configuration of internal switches in a crossbar

Shared memory interconnects - Switched

- For example, the Figure below shows the configuration of the switches if **P1** writes to **M4**, **P2** reads from **M3**, **P3** reads from **M1**, and **P4** writes to **M2**.



simultaneous memory accesses by the processors

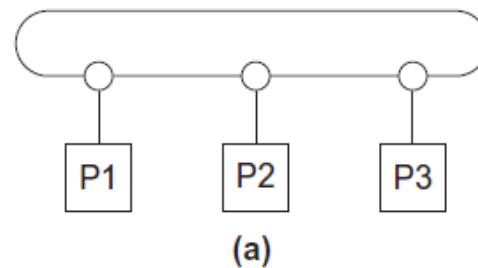
Distributed memory interconnects

- Two groups
 1. Direct interconnect
 - Each switch is directly connected to a processor-memory pair, and the switches are connected to each other.
 - **Example: ring** and a two-dimensional **toroidal mesh**
 2. Indirect interconnect
 - Switches may not be directly connected to a processor.
 - **Example: Crossbar - omega network**

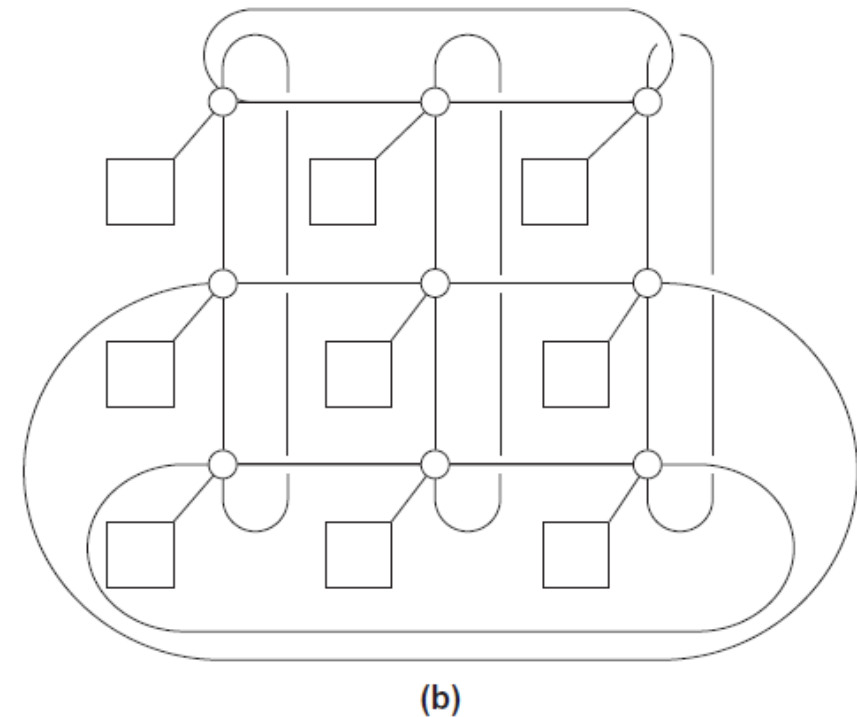
Direct interconnect

Figure 2.8 shows a ring and a two-dimensional toroidal mesh. As before, the **circles** are switches, the **squares** are processors, and the lines are **bidirectional** links.

- **Ring:** Each switch has 3 links;
 - p processors = p links.
- **Toroidal mesh:** Each switch has 5 links;
 - P processors = $2p$ links.



ring



toroidal mesh

Direct interconnect

- One of the simplest measures of the power of a direct interconnect is the number of links.
 - When counting links in a direct interconnect, it's customary to count only **switch-to-switch** links. This is because the speed of the processor-to-switch links may be very different from the speed of the switch-to-switch links.
- To get the total number of links, we can usually just add the number of processors to the number of switch-to-switch links.
 - So, in the diagram for a ring (Fig. 2.8a), we would ordinarily count **3 links** instead of 6, and in the diagram for the toroidal mesh (Fig. 2.8b), we would count **18 links** instead of 27.

Direct interconnect - Ring

- (+) A ring is superior to a simple bus since it **allows multiple simultaneous communications**.
- (-) However, it's easy to devise communication schemes, in which some of the processors must wait for other processors to complete their communications.

Direct interconnect - Toroidal mesh

- (+) The number of possible **simultaneous communication patterns** is **greater** with a mesh than with a ring.
- (-) The toroidal mesh will be **more expensive** than the ring because the switches are more complex—they must support five links instead of three—and if there are p processors, the number of links is $2p$ in a toroidal mesh, while it's only p in a ring.

Hypercube

- Highly connected direct interconnect that has been used in actual systems.
- 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. (Each processor has direct connections to two neighbors)
 - Similarly, a **three-dimensional hypercube** is built from two two-dimensional hypercubes. (Each processor has direct connections to three neighbors)

Hypercubes

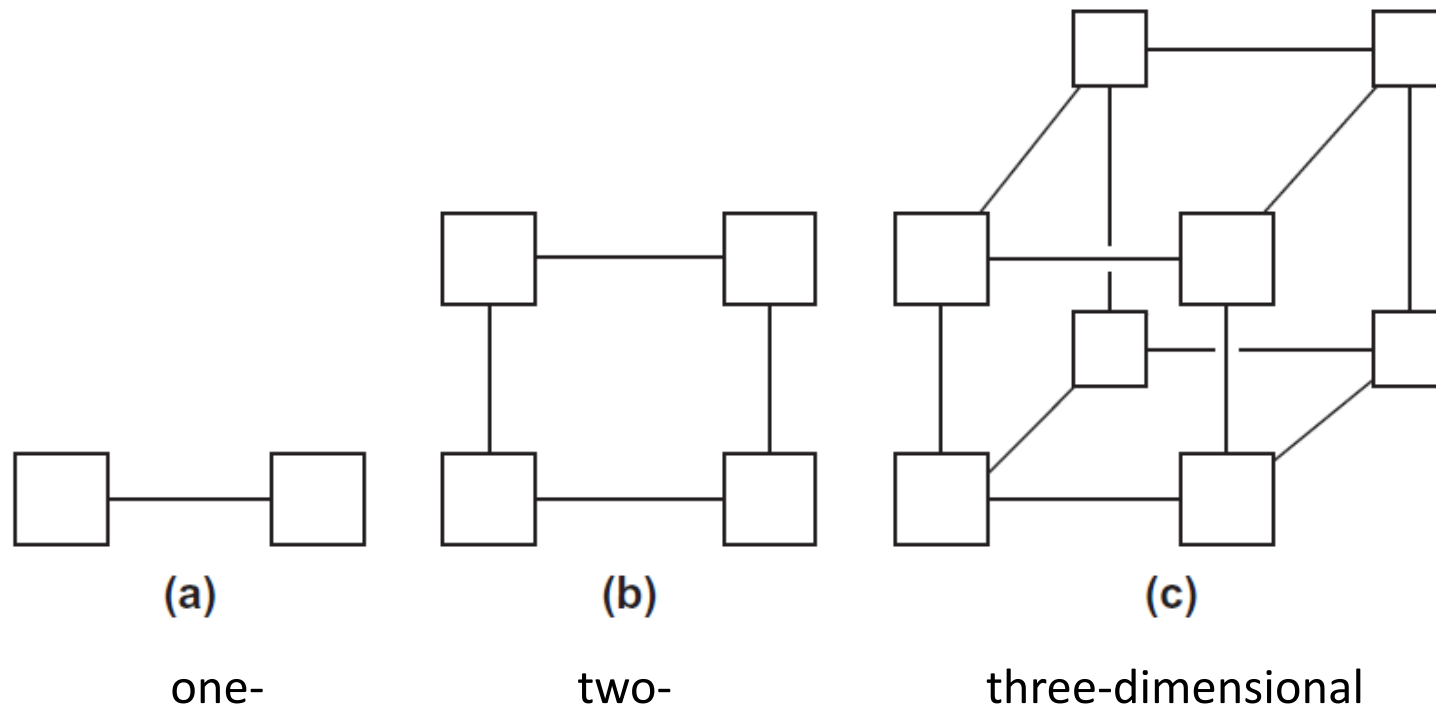


Figure 2.12

Indirect interconnects

- In an indirect interconnect, the switches may not be directly connected to a processor.
- Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.
- Simple examples of indirect networks:
 - Crossbar
 - Omega network

A generic indirect network

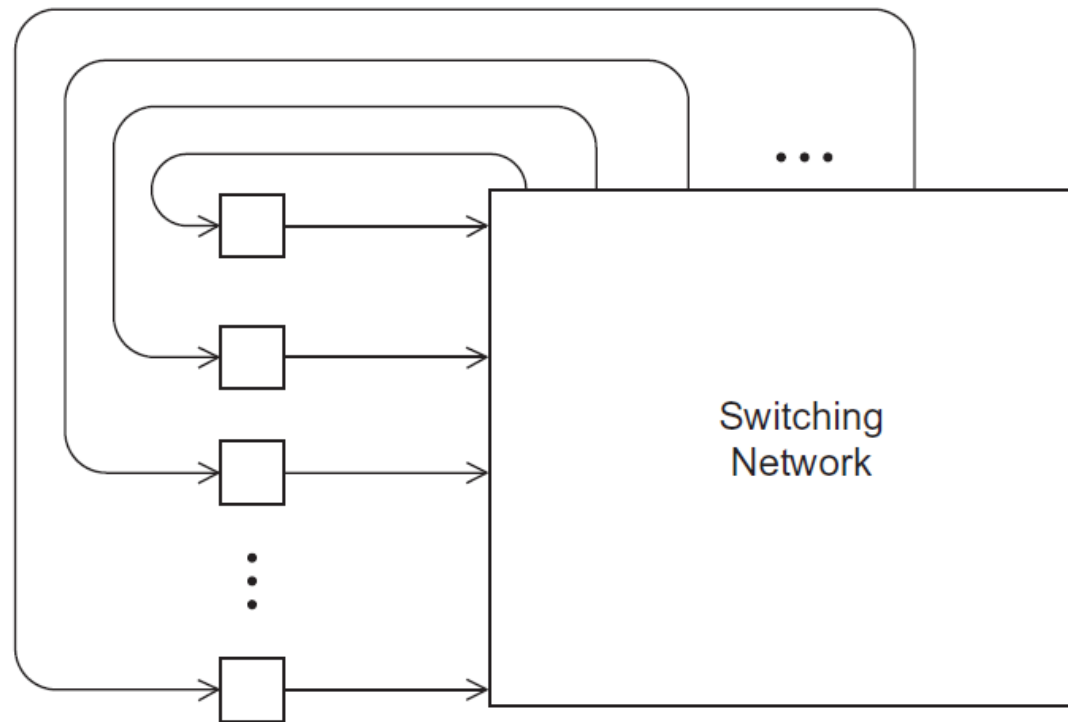


Figure 2.13

Crossbar interconnect for distributed memory

- The diagram of a distributed-memory crossbar in Fig. 2.14 has unidirectional links.
- Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.

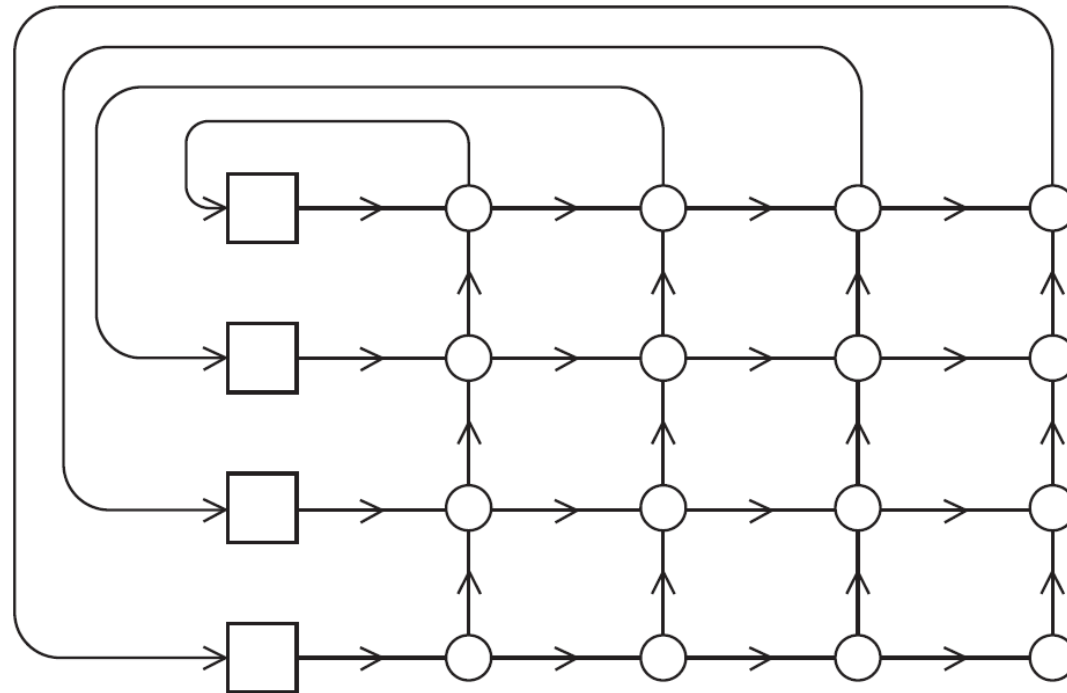


Figure 2.14

An omega network

- An omega network is shown in Fig. 2.15. The switches are two-by-two crossbars (see Fig. 2.16).
- Observe that, unlike the crossbar, there are communications that cannot occur simultaneously.
- For example, in Fig. 2.15, if processor 0 sends a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7.
- On the other hand, the omega network is less expensive than the crossbar. It uses a smaller number of switches.

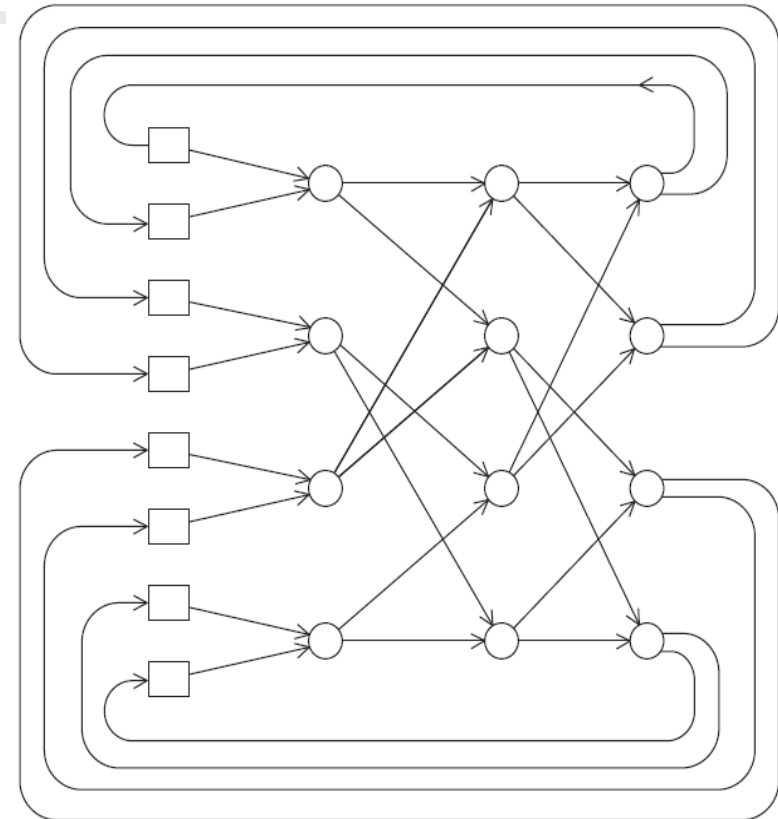


Figure 2.15

A switch in an omega network

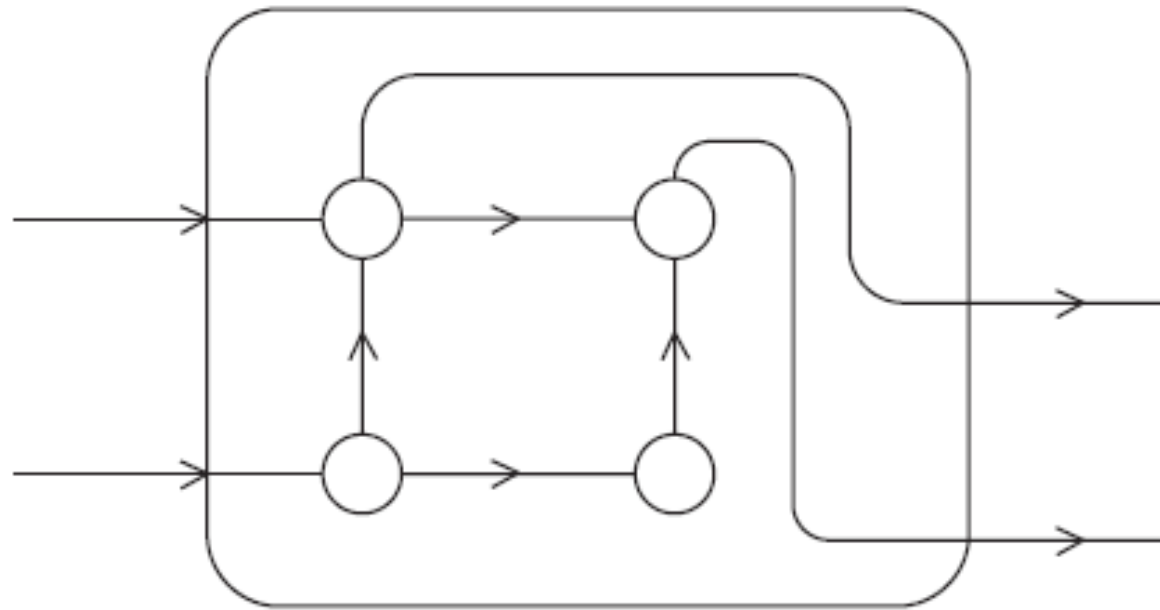
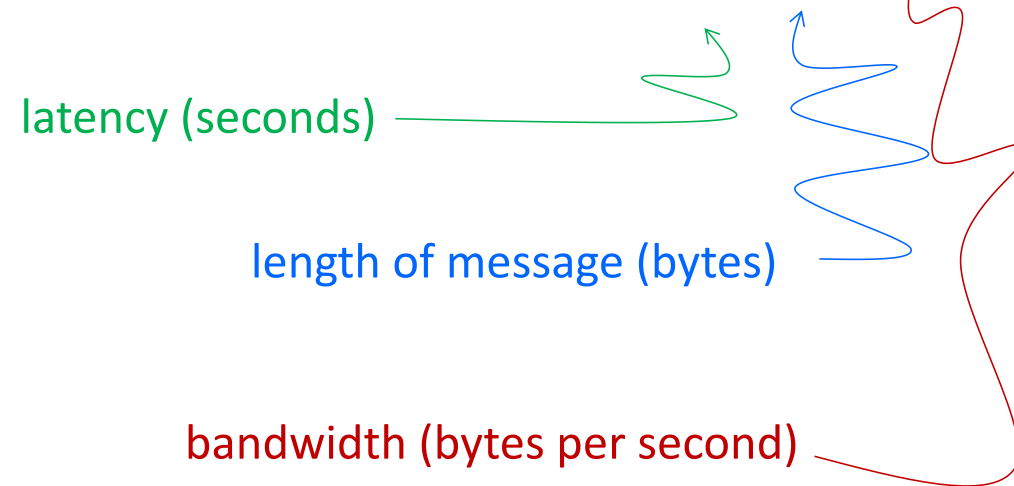


Figure 2.16

More definitions

- Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.
- This is true whether we're talking about transmitting data between **main memory and cache**, **cache and register**, **hard disk and memory**, or between **two nodes in a distributed-memory or hybrid system**.
- There are two figures that are often used to describe the performance of an interconnect (regardless of what it's connecting):
 - **Latency**: The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.(e.g., 200ms)
 - **Bandwidth**: Amount of data that can be transferred over a period of time (e.g., MB/sec).
- So, if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is message transmission time = $l + n/b$.

Message transmission time = $l + n / b$



Basics of caching

- A collection of memory locations that can be accessed in **less time** than some other memory locations.
- By Caches we mean a **CPU cache**
- A CPU cache is a collection of memory locations that the CPU can access **more quickly** than it can access main memory.
- A CPU cache can either be:
 - located on the same chip as the CPU or
 - it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.



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.

Principle of locality

- Which data and instructions should be stored in the cache?
 - Idea: programs tend to **use data and instructions** that are **physically close to recently used** data and instructions.

```
float z[1000];
```

```
sum = 0.0;
```

```
for (i = 0; i < 1000; i++)
```

```
    sum += z[i];
```

- **Arrays are allocated as blocks of contiguous memory locations.** So, for example, the location storing `z[1]` immediately follows the location `z[0]`. Thus as long as `i < 999`, the **read of `z[i]`** is immediately followed by a **read of `z[i+1]`**.
- The principle that **an access of one location** is followed by **an access of a nearby location** is often called **locality**.

Cache Terminology

- **Cache levels:** cache is usually divided into levels: the first level (L1) is the smallest and the fastest, and higher levels (L2, L3, . . .) are larger and slower.
 - Most systems currently, have **at least two levels** and having three levels is quite common.
 - **When the CPU needs to access instruction or data**, it works its way down the cache hierarchy: **First**, it checks the **level 1** cache, then the **level 2**, and so on. **Finally**, if the information needed isn't in any of the caches, it accesses the **main memory**.
- **Cache hit:** requested value found in the cache.
- **Cache miss:** value not found, request passed up to next cache, or interconnect
 - Hit or miss is often modified by the level. For example, when the CPU attempts to access a variable, it might have an L1 miss and an L2 hit.
- **Cache line:** The cache is split into lines; each line is multiple bytes (64-256 bytes). A memory request will obtain multiple bytes.

Levels of Cache

smallest & fastest

L1

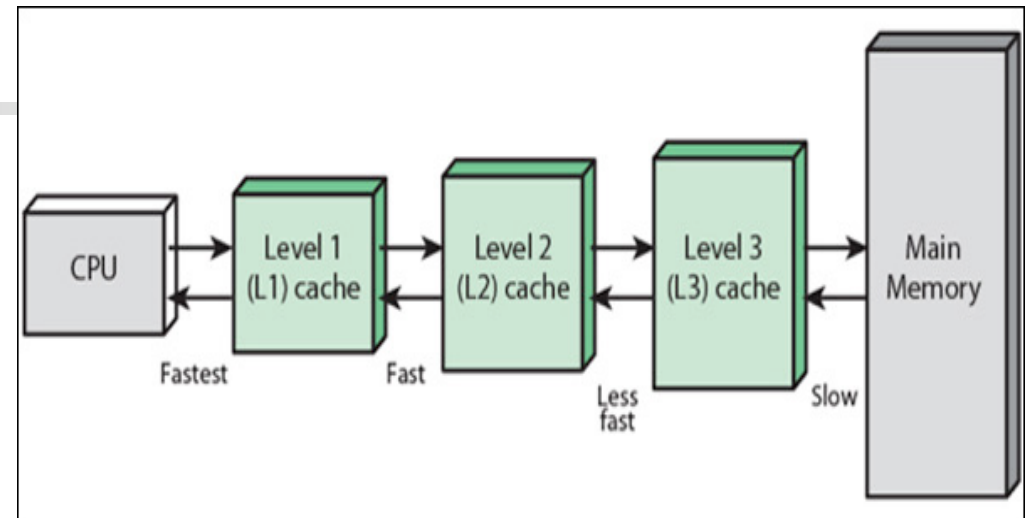


L2

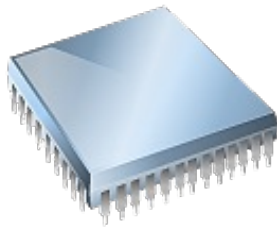
L3



largest & slowest



Cache hit



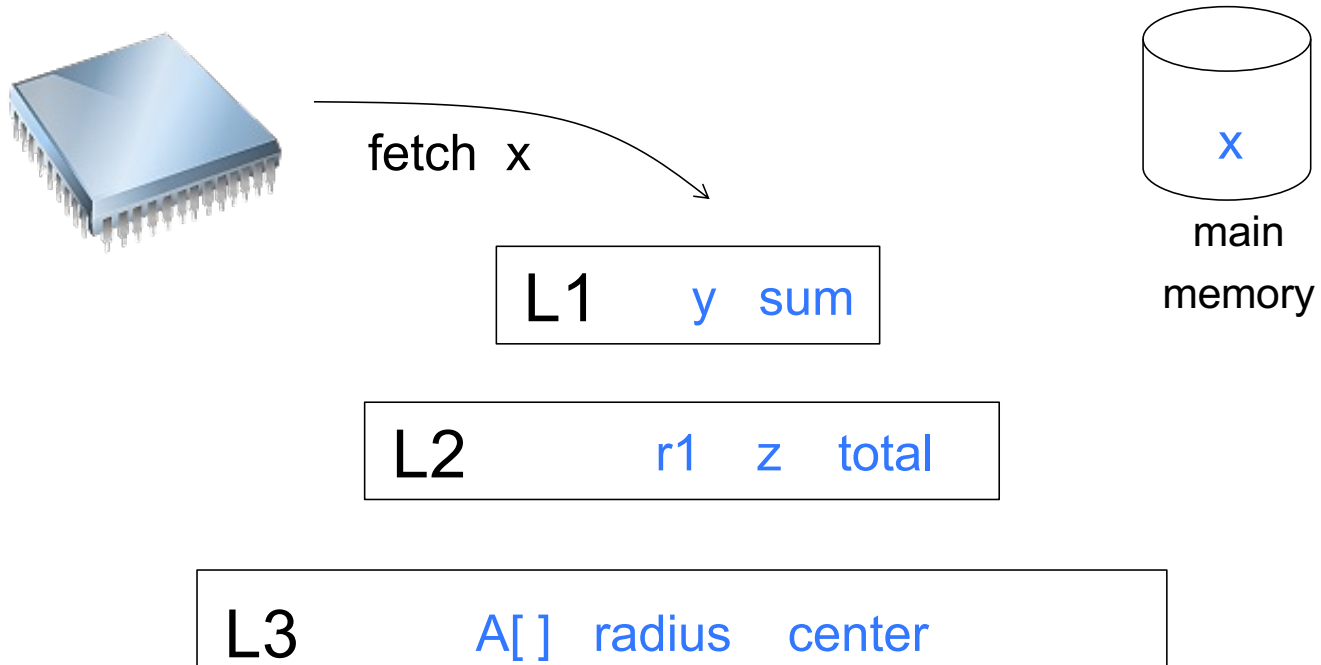
fetch x

L1 x sum

L2 y z total

L3 A[] radius r1 center

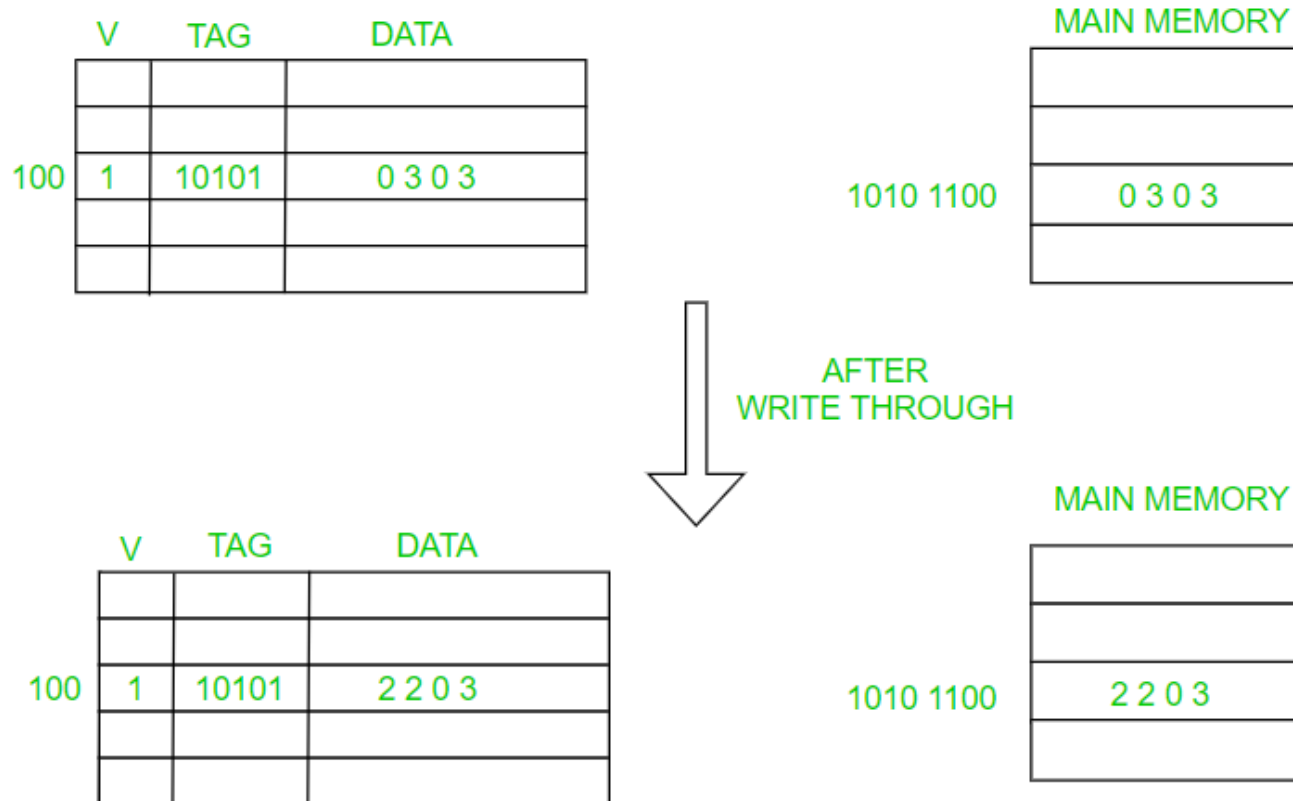
Cache miss



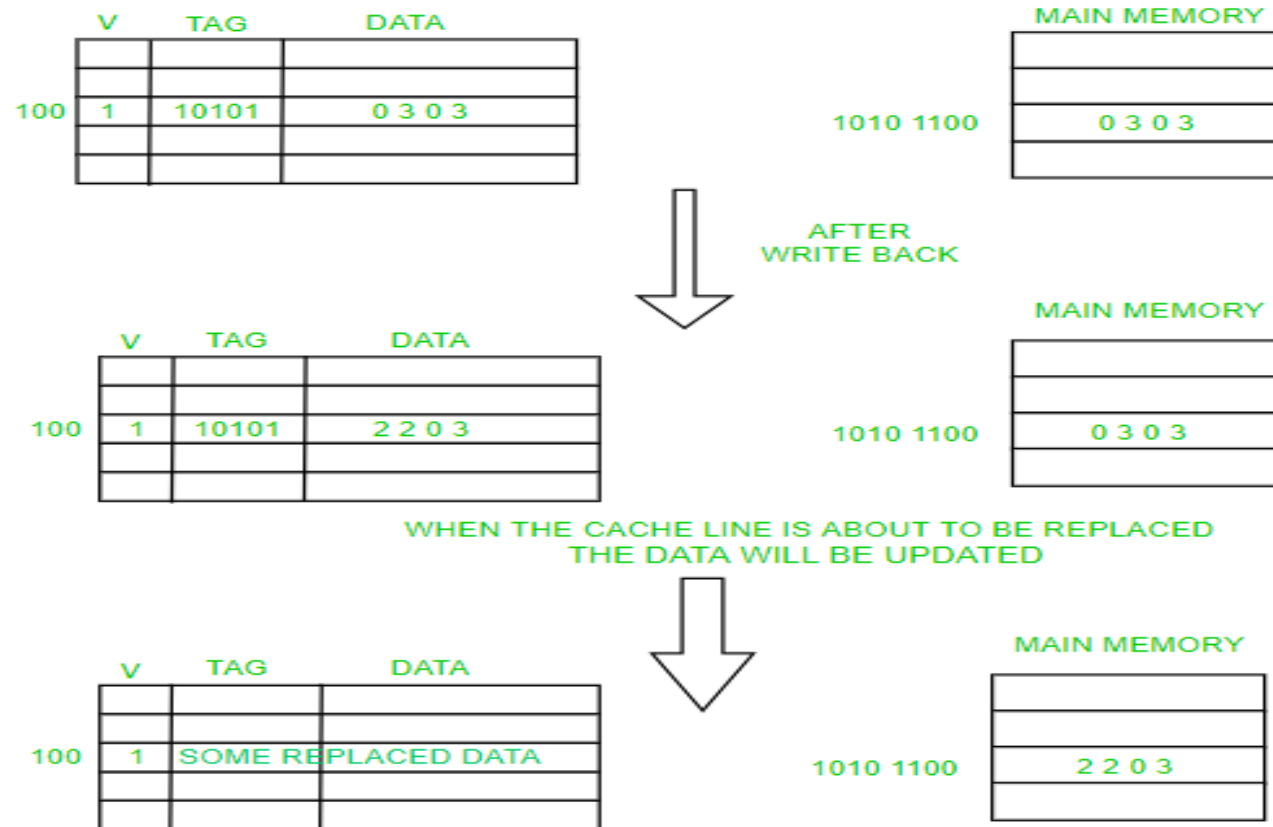
Issues with cache -- Write Strategy

- When a CPU writes data to the cache, the value in the cache may be inconsistent with the value in main memory.
- There are two basic approaches to dealing with inconsistency.
 1. **Write-through** caches handle this by updating the data in the main memory at the time it is written to the cache.
 2. **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.

Issues with cache – Write-Through



Issues with cache – Write-Back



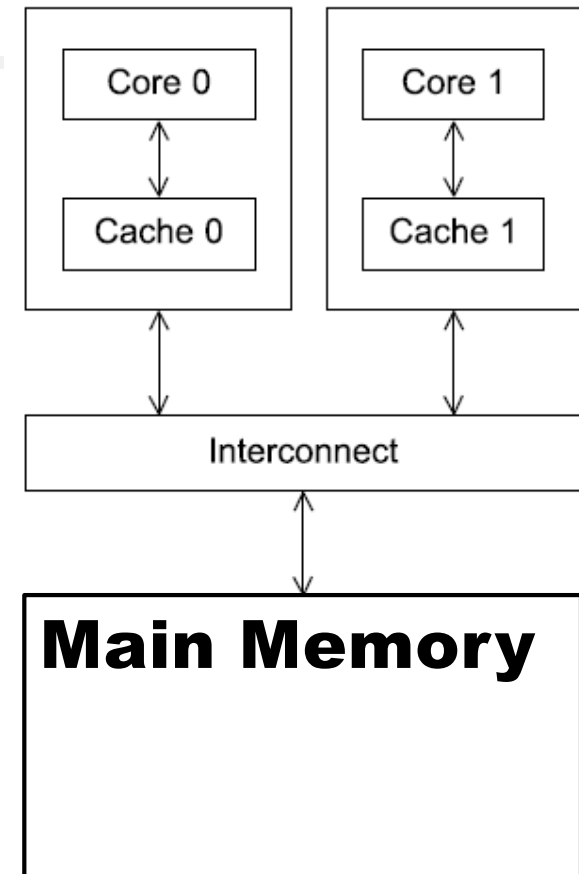
Cache coherence

- Recall that CPU caches are managed by system hardware.
- Programmers have no control over caches and when they get updated → This has several important consequences for shared-memory systems.
- https://www.youtube.com/watch?v=r_ZE1XVT8Ao&list=PLBlnK6fEyqRgLLIzdgiTUKULKJPYc0A4q&index=29

Cache coherence

- Assume just single-level caches and main memory.
- Processor writes to the location in its cache
- Other caches may hold shared copies -these will be out-of-date.
- Updating main memory alone is not enough

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

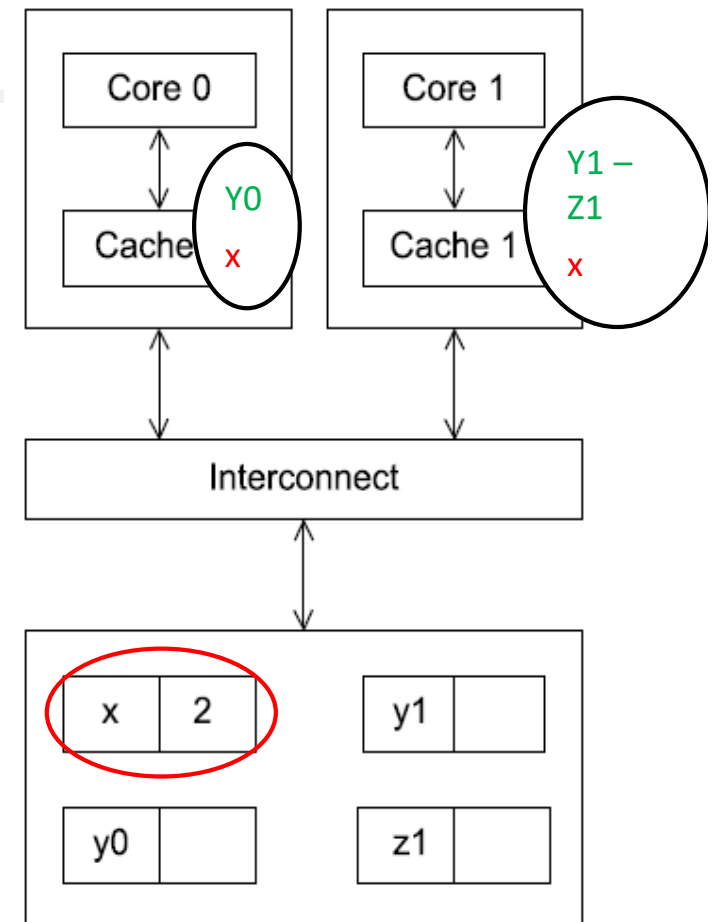
x = 2; /* shared variable */

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 = ???



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 = ???

z1? its value $4 \times 7 = 28$.

z1? its value $4 \times 2 = 8$.

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy.

Write-through policy: The main memory will be updated by the assignment $x = 7$. However, this will **have no effect on the value in the cache of core 1**.

Write-back policy: The **new value of x** in the cache of core 0 probably **won't even be available to core 1** when it updates z1.

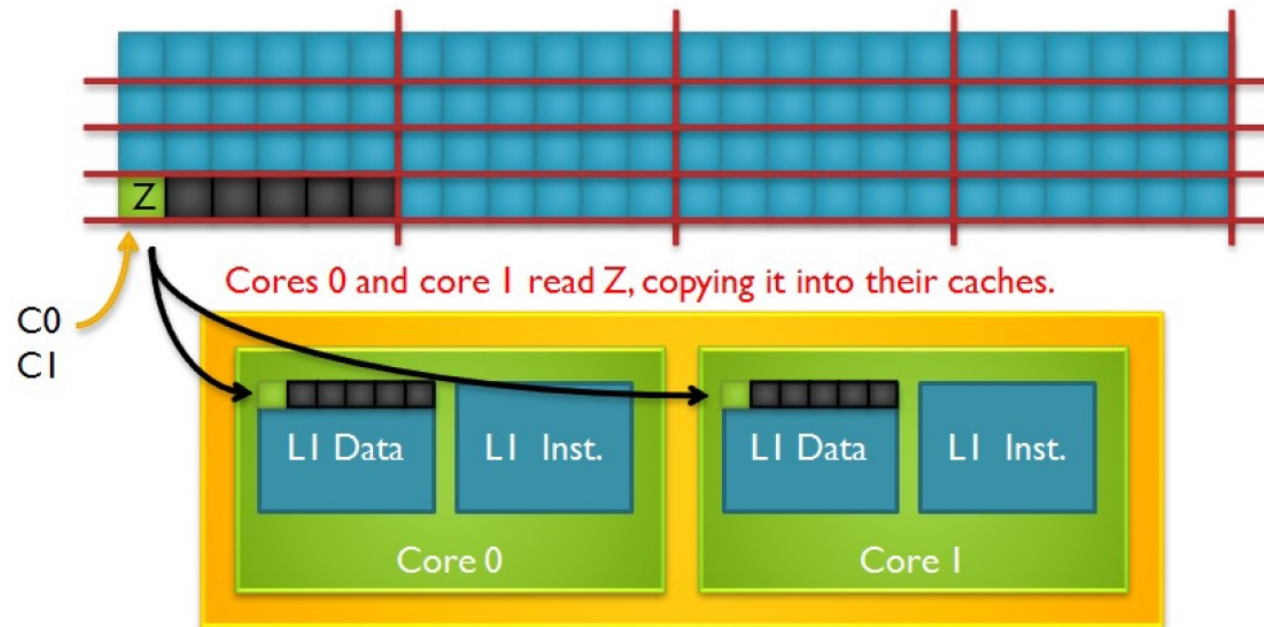
Two different copies of the same information contain different values
→ **Cache Coherence Problem**

Caches for Multi-core CPUs

- **Require special treatment**
 - Core **reads/writes** variables in its L1 cache
 - If two cores have the **same variable in L1**, then **they need some way to make sure they both see the same value for that variable.**
- This is called **cache coherency**. It makes sure both processors see the same **(latest) value**.

Cache Coherency Details

Example: Need Cache Coherency:



If Core 0 changes the value of Z, core 1 needs to know about it

Cache Coherence

- To ensure coherency, some information is maintained within the cache controller – A **state** is specified for each cache line:
 1. **Modified**: This indicates that the cache line is present in the current cache only and is dirty i.e., its value is different from the main memory. The cache is required to write the data back to the main memory in the future, before permitting any other read of the invalid data in the main memory.
 2. **Exclusive** – This indicates that the cache line is present in the current cache only and is clean i.e., its value matches the main memory value.
 3. **Shared** – It indicates that this cache line may be stored in other caches of the machine. These copies are considered "clean" as they match the main memory.
 4. **Invalid** – It indicates that this cache line is invalid.

Cache Coherence

- There are two main approaches to ensuring cache coherence:
 1. **snooping cache coherence**
 1. Write Invalidate
 2. Write Update
 2. **directory-based cache coherence**

Snooping Cache Coherence

- The idea behind snooping comes from **bus-based systems**: The cores share a bus.
- Any signal transmitted on the bus can be “**seen**” by all cores connected to the bus.
- Each CPU (cache system) ‘snoops’ (i.e. watches continually) for write activity concerned with data addresses that it has cached.
- Two main snooping protocols:
 1. **Write-update** protocol
 2. **Write-invalidate** protocol
- Both protocols aim to maintain cache coherence and ensure that processors work with consistent data, but they have different trade-offs in terms of bus bandwidth and latency.

Snooping Cache Coherence

- When core 0 updates the copy of x stored in its cache it also **broadcasts** this information across the bus, as (**write invalidate** or **write update**) message.
- 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 (**Write invalidate**) or update its copy (**Write Update**).
 - **Write invalidate** protocol reduces unnecessary data transfer but can lead to increased memory access (latency) when a processor needs the updated data.
 - **Write Update** protocol eliminates the latency associated with invalidations, it can generate more bus traffic due to the frequent updates, which may impact system performance.

<https://www.youtube.com/watch?v=YNpaELJZm2c&list=PLBlnK6fEyqRgLLIzdgiTUKULKJPYc0A4q&index=30>

Snooping Cache Coherence

- it's not essential that the interconnect be a bus, only that it supports broadcasts from each processor to all the other processors.
- **snooping** works with both **write-through** and **write-back** caches.
 - Write-through: modification propagated to main memory immediately.
 - Write-back: modification propagated to main memory during the next cache replacement.

Snooping Cache Coherence

- (+) **Simple**, tends to be **faster** if enough bandwidth is available since all requests/responses are seen by all cores.
- (-) In larger systems such as NUMA, broadcasts are **expensive**, and snooping cache coherence **requires a broadcast every time a variable is updated**. In such systems, broadcasts across the interconnect will be **very slow** relative to the speed of accessing local memory.
- (-) So snooping cache coherence **isn't scalable**, because for larger systems it will cause **performance to degrade**.

Directory Based Cache Coherence

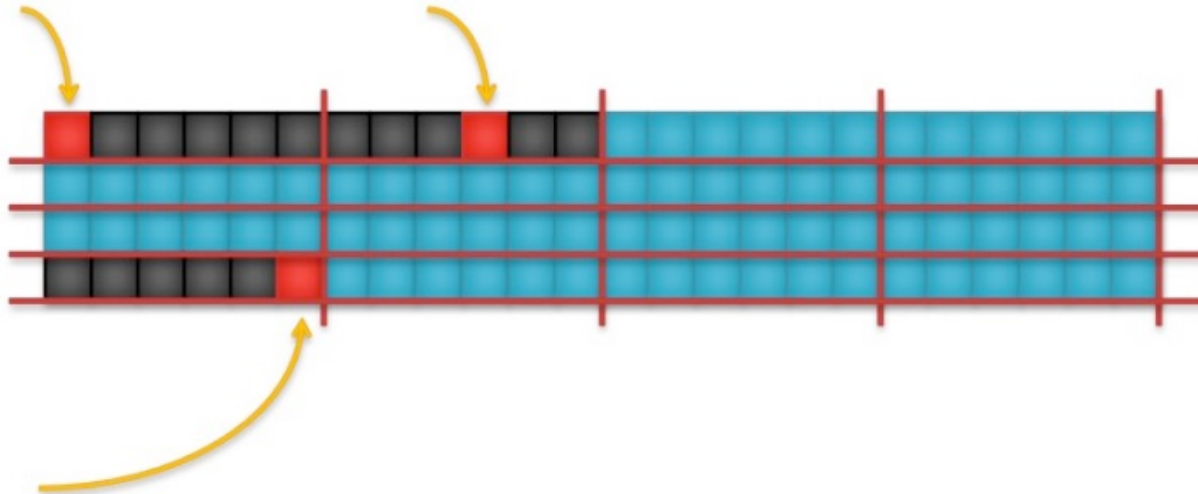
- Uses a data structure called a **directory** that stores the **status of each cache line**.
- Typically, this data structure is **distributed**; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory.
- Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated, indicating that core 0 has a copy of the 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 **will invalidate those lines**.
 - (-) Clearly, there will be **substantial additional storage** required for the directory.
 - (+) BUT, when a cache variable is updated, only the cores storing that variable need to be contacted.
- <https://www.youtube.com/watch?v=KEc4NQZjkMI&list=PLBlnK6fEyqRgLLIzdgiTUKULKJPYc0A4q&index=31>

False sharing

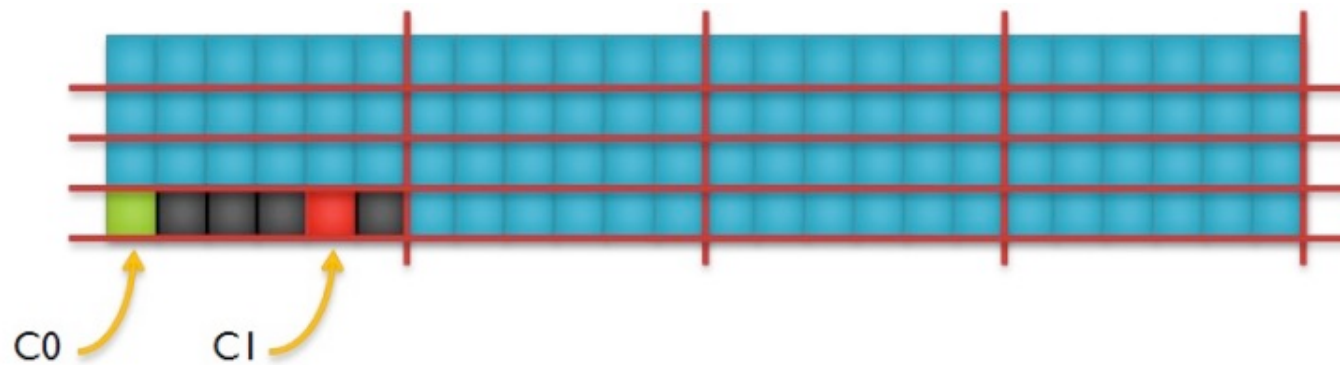
- It's important to remember that CPU caches are implemented in hardware, so they **operate on cache lines**, not individual variables.
- This can have disastrous consequences for performance.

False sharing

- Each **cache line** holds multiple bytes
 - 64-256 bytes per line
 - To read/write one byte, **need a whole line**



Example(1): False Sharing X and Y



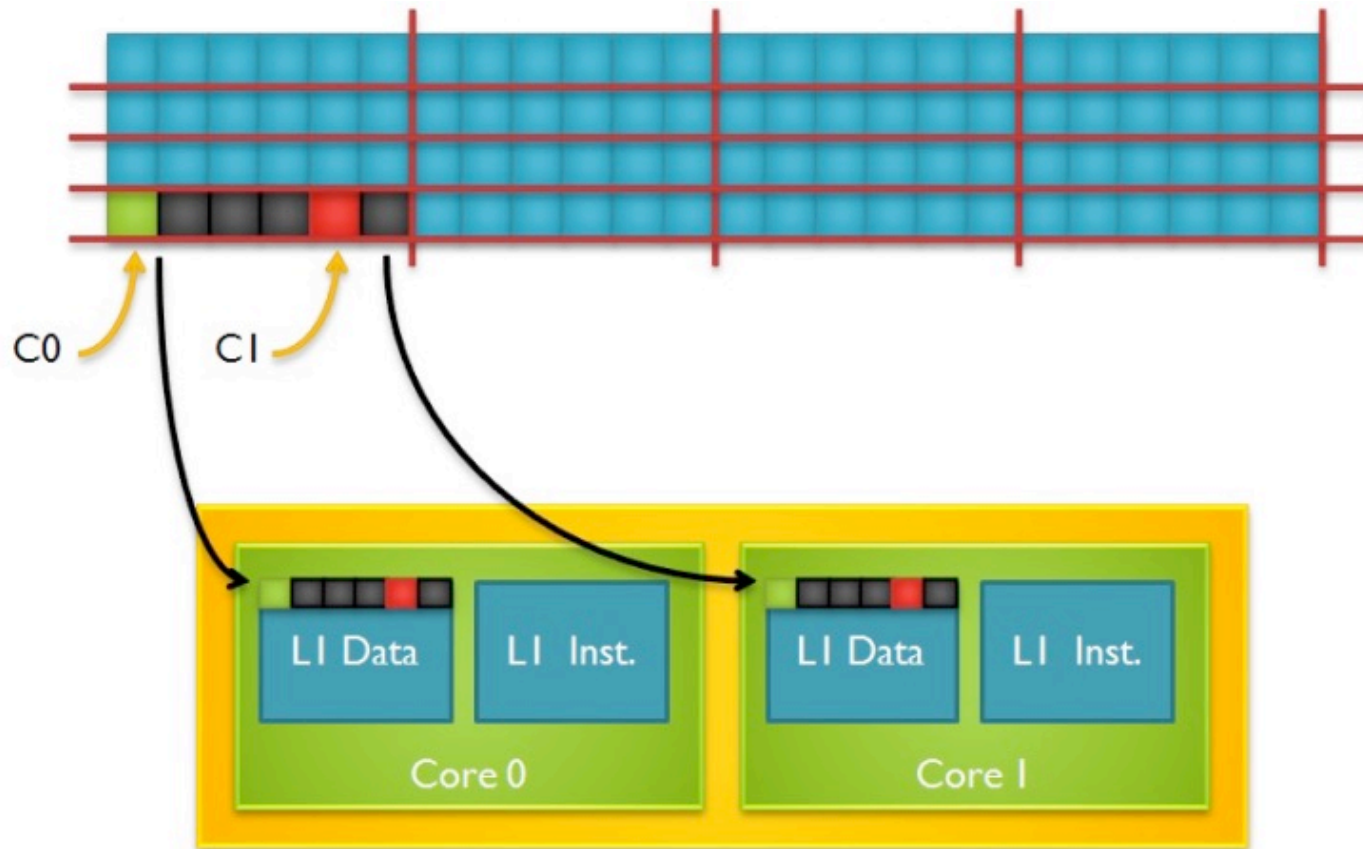
■ False sharing

- Core 0 wants to change value X
- Core 1 wants to change value Y
- **X and Y are different values**, with **different addresses in memory**
- But if they are **near each other in memory**, they could fit in the same cache line

Example(1): False Sharing X and Y

- False sharing
 - Core 0 wants to change value X
 - Core 1 wants to change value Y
 - If X and Y are in the same cache line, then both cores need **exclusive access** to that line.
 - Only one core can have exclusive access
 - The other core has to wait
 - Wastes time, reduces performance
 - It cannot execute in parallel
 - This is **false sharing**. X and Y are different values with different memory addresses. But because they are near, they fit in the same cache line, requiring the cores to share the line.

Example(1): False Sharing X and Y



Example(2): False Sharing Array Sum

- Another example, suppose we want to repeatedly call a function $f(i, j)$ and add the computed values into a **vector**:
- We **can parallelize this by dividing the iterations in the outer loop among the cores.**
- If we have *core_count* cores, we might assign the first $m/\text{core_count}$ iterations to the **first core**, the next $m/\text{core_count}$ iterations to the **second core**, and so on.

Example(2): False Sharing Array Sum

```
/ Private variables /  
int i, j, iter_count;  
/ Shared variables initialized by one core /  
int m, n, core_count  
double y[m];  
iter_count = m/core_count  
/ Core 0 does this /  
for (i = 0; i < iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);  
/ Core 1 does this /  
for (i = iter-count+1; i < 2*iter_count; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);  
...
```

Example(2): False Sharing Array Sum

- Now suppose our shared-memory system has two cores, $m = 8$,
- Since **doubles = 8 bytes**, cache lines are $8 \times 8 = 64$ bytes, and $y[0]$ is stored at the beginning of a cache line.
- A **cache line can store eight doubles**, and **y takes one full cache line**.
- What happens when **core 0 and core 1 simultaneously** execute their codes?
 - Since all of y is stored in a single cache line, each time one of the cores executes the statement $y[i] += f(i,j)$, **the line will be invalidated**, and the next time the other core tries to execute this statement it will have to **fetch the updated line from memory!**
 - So if n is large, we would expect that a large percentage of the assignments $y[i] += f(i,j)$ will access main memory—in spite of the fact that core 0 and core 1 **never access each others' elements** of y .

Example(2): False Sharing Array Sum

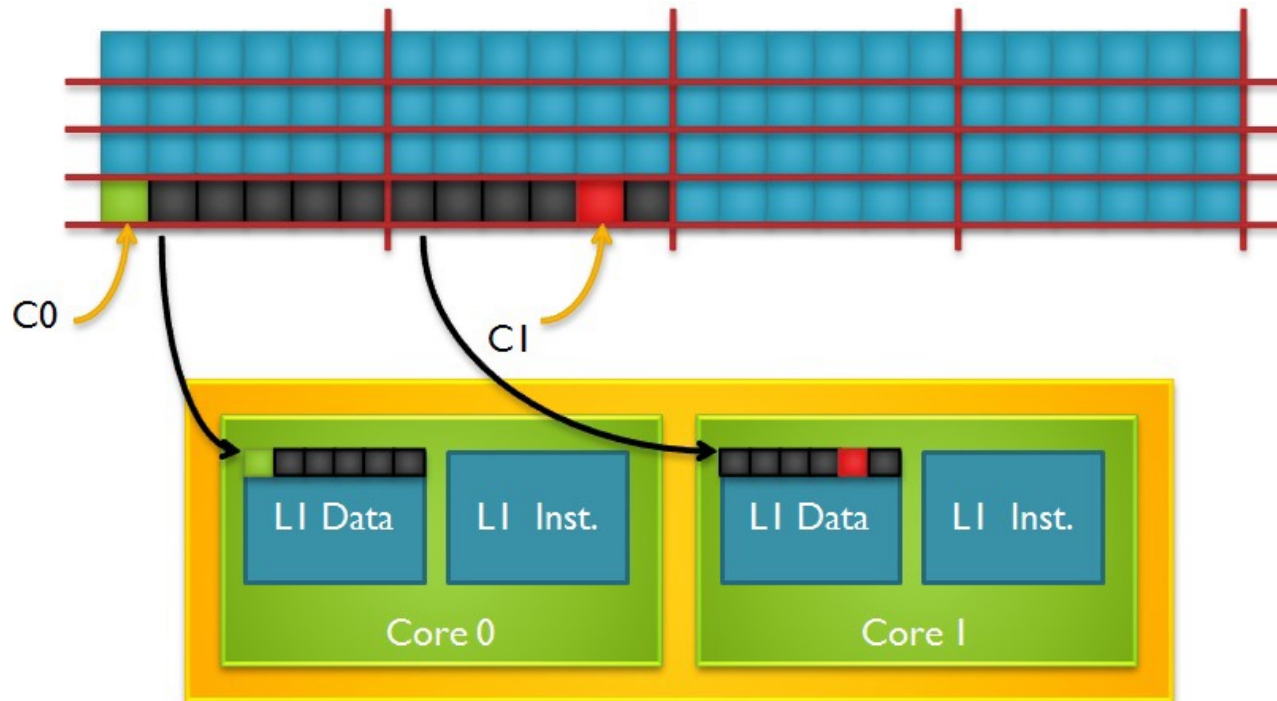
- This is called **false sharing**, because the system is behaving *as if* the elements of **y** were being shared by the cores.
- Note that false sharing **does not cause incorrect results**. However, it can **ruin** the **performance** of a program **by causing many more accesses** to memory than necessary.

Removing False Sharing

- False sharing can be reduced using several techniques such as:
 1. **Padding**: Add padding (unused variables or arrays) between the variables that are frequently accessed by different threads. This ensures that they reside in separate cache lines.
 2. Make use of **temporary storage** that is local to the **thread** or **process** and then copy the temporary storage to the shared storage.
 3. utilizing the compiler's optimization features to eliminate memory loads and stores.
 - Enforce padding
 - Restructure loops to reduce the chances of threads accessing the same cache line.
 - Compiler-specific flags (e.g., -O2, -O3 in GCC) enable various optimizations
 - Some compilers provide directives or hints (e.g., `__attribute__((aligned))`) in GCC that specify the alignment requirements for variables or data structures.

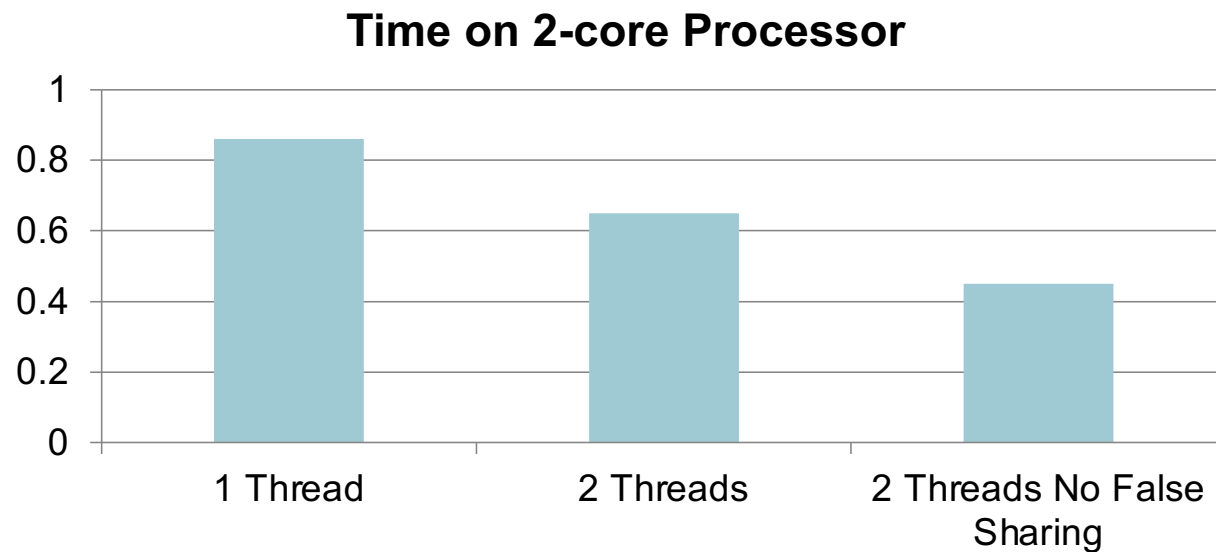
Removing False Sharing

- **Padding** to move X and Y to different cache lines



Performance of Array Sum

- With **false sharing** removed



- Study the **Cache coherency and false sharing**, watch the following video:

<https://www.youtube.com/watch?v=dznxqe1Uk3E>

Summary

■ Classification of computer architectures: Flynn taxonomy

1. **SISD** – Von-Neuman
2. **SIMD** – Vector processors - GPUs
3. **MISD** – NA
4. **MIMD** – Two main categories:
 1. Shared Memory: UMA – NUMA.
 2. Distributed Memory: Cluster – Grid.

■ Interconnection networks

1. Shared memory interconnect: Bus – Switched.
2. Distributed memory interconnect:
 - Direct: Ring – Toroidal Mesh – Hypercube.
 - Indirect: Crossbar – Omega network.

■ Cache coherency

- Snooping cache coherence
- Directory-based cache coherence

■ False sharing