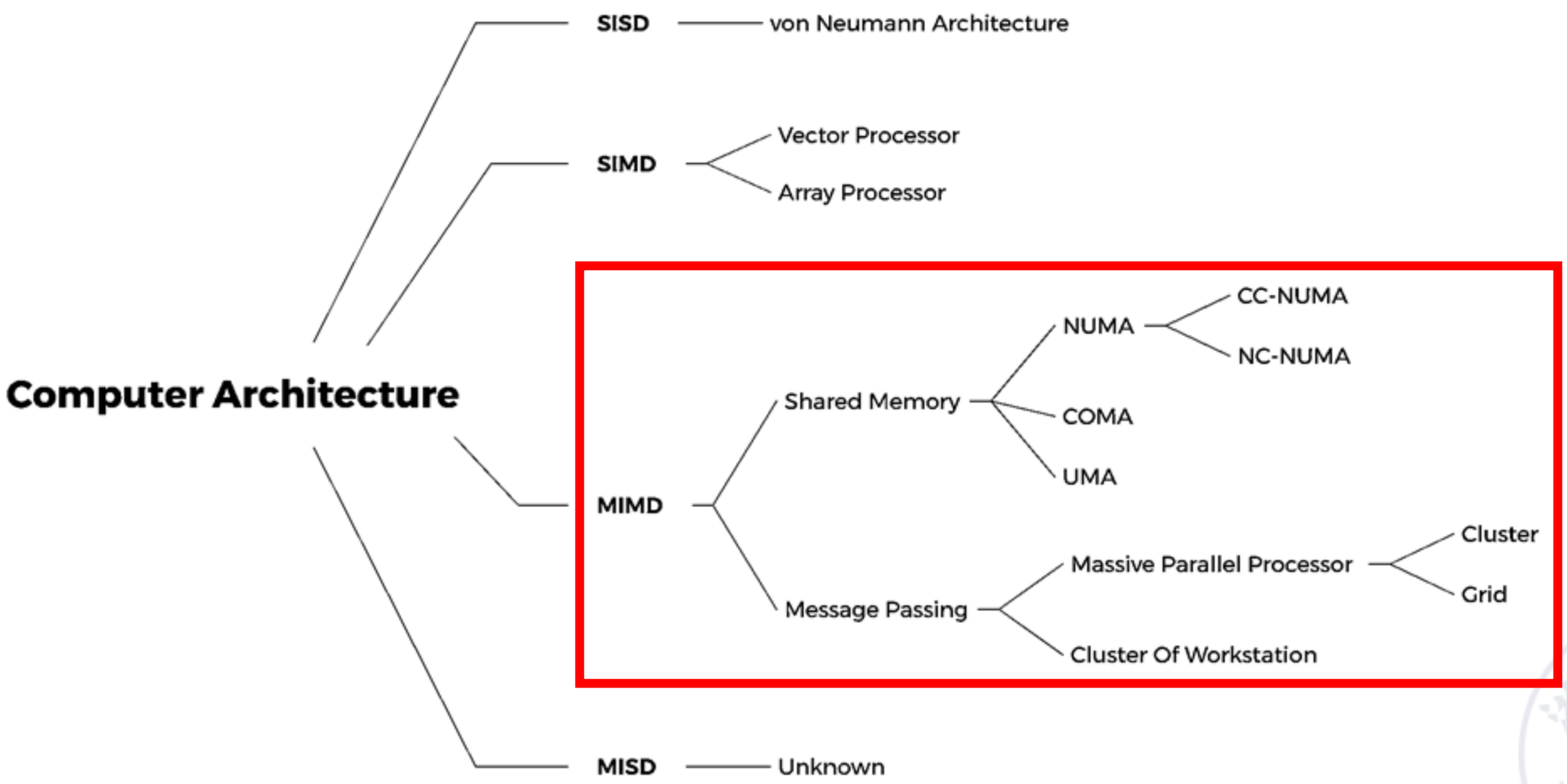
Chapter 5

DLP and TLP

Flynn


Computer Architecture
SISD
von Neumann Architecture
SIMD
Vector Processor
Array Processor
MIMD
Shared Memory
NUMA
CC-NUMA
NC-NUMA
COMA
UMA
Message Passing
Massive Parallel Processor
Cluster
Grid
Cluster Of Workstation
MISD
Unknown

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

W. Jack Bouknight et al.,
The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel's future direction at the Intel Developer Forum in 2005

Since 2004 processor designers have increased core counts to exploit Moore's Law scaling, rather than focusing on single-core performance. The failure of Dennard scaling, to which the shift to multicore parts is partially a response, may soon limit multicore scaling just as single-core scaling has been curtailed.

Hadi Esmaeilzadeh, et al.,
Power Limitations and Dark Silicon Challenge the Future of Multicore (2012)

From ILP to TLP

- Thread-level parallelism is identified at a high level by software system or programmer;
- The threads consist of hundreds to millions of instructions that may be executed in parallel.

From TLP to MIMD Architecture

- TLP implies the existence of multiple program counters.
- Thus TLP is exploited primarily through MIMDs

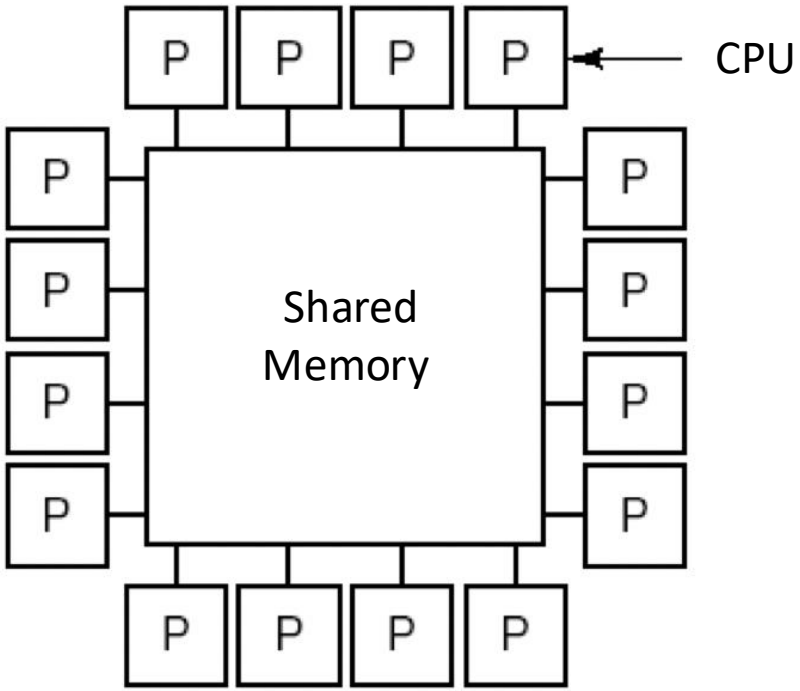
MIMD Architecture

- Multi-processor system——based on shared memory
  - There is only a unique address space in the system, and all processors share this address space.
  - A unique address space does not mean that there is only one memory physically. The shared address space can be realized by a physically shared memory, or can be realized by a distributed memory with the support of hardware and software.
- Multi-computer system——based on message passing
  - Each processor has its own memory, which can only be accessed by the processor and cannot be directly accessed by other processors. This type of memory is called local memory or private memory.
  - When processor A needs to send data to processor B, A sends the data to B in the form of a message.

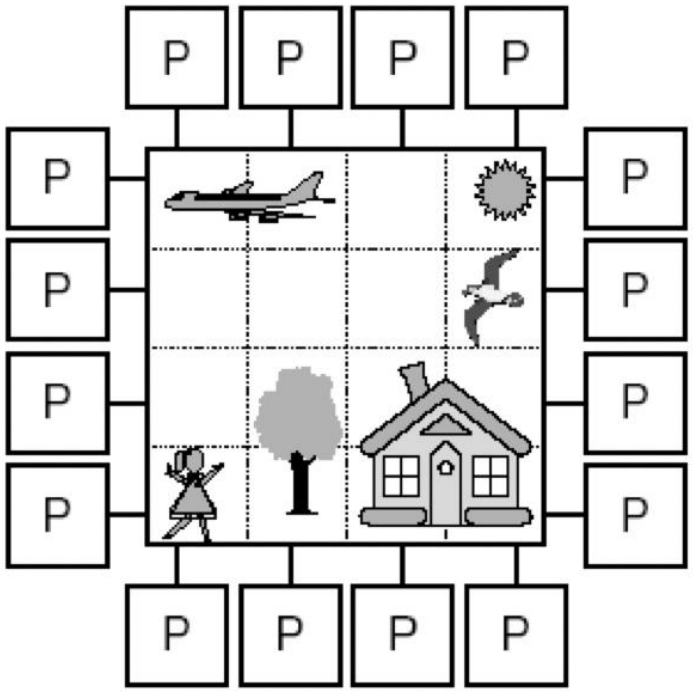
MIMD Architecture

- Multi-processor system——based on shared memory
  - There is only a unique address space in the system, and all processors share this address space.
  - A unique address space does not mean that there is only one memory physically. The shared address space can be realized by a physically shared memory, or can be realized by a distributed memory with the support of hardware and software.
- Multi-computer system——based on message passing
  - Each processor has its own memory, which can only be accessed by the processor and cannot be directly accessed by other processors. This type of memory is called local memory or private memory.
  - When processor A needs to send data to processor B, A sends the data to B in the form of a message.

Multi-process system based on Shared Memory



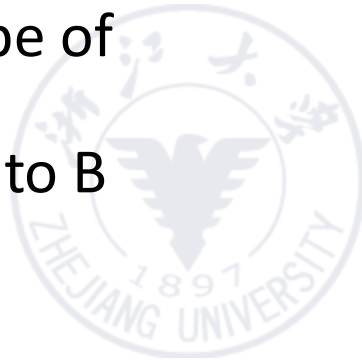
(a) A shared memory system with 16 CPUs



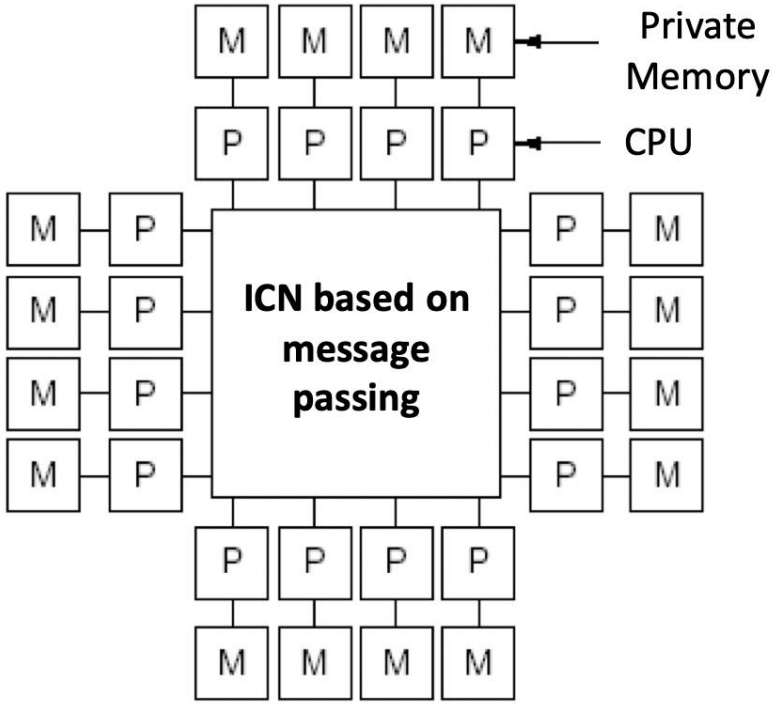
(b) A pictures is divided into 16 parts which are processed by different CPUs

MIMD Architecture

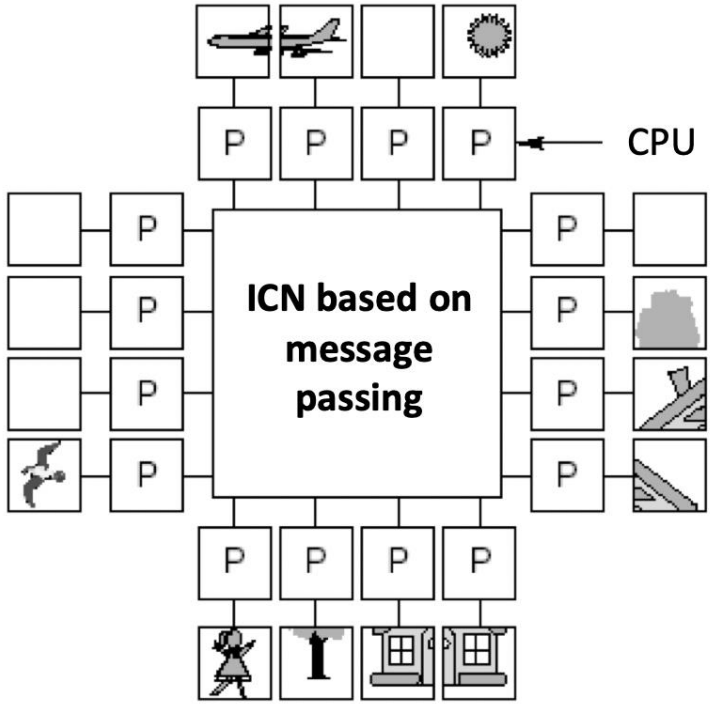
- Multi-processor system——based on shared memory
  - There is only a unique address space in the system, and all processors share this address space.
  - A unique address space does not mean that there is only one memory physically. The shared address space can be realized by a physically shared memory, or can be realized by a distributed memory with the support of hardware and software.
- Multi-computer system——based on message passing
  - Each processor has its own memory, which can only be accessed by the processor and cannot be directly accessed by other processors. This type of memory is called local memory or private memory.
  - When processor A needs to send data to processor B, A sends the data to B in the form of a message.



Multi-computer system based on message passing



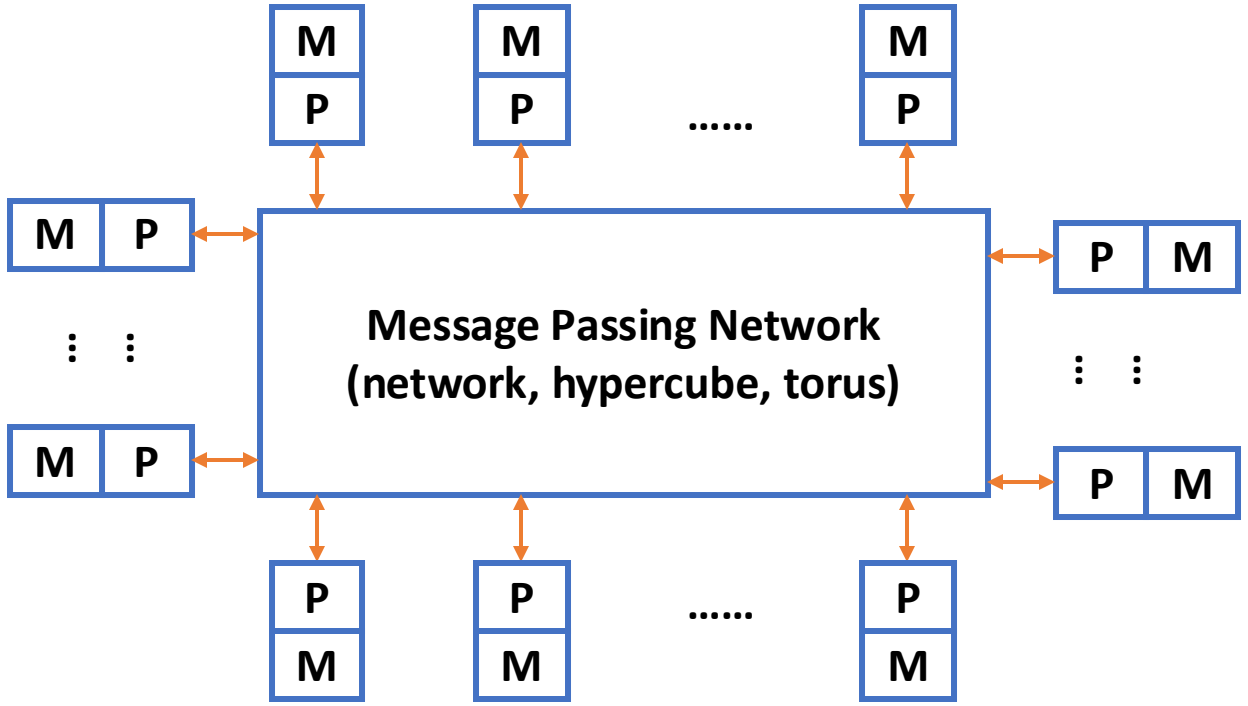
(a) Every CPU owns its private memory



(b) A pictures is divided into 16 parts which are processed and stored separately

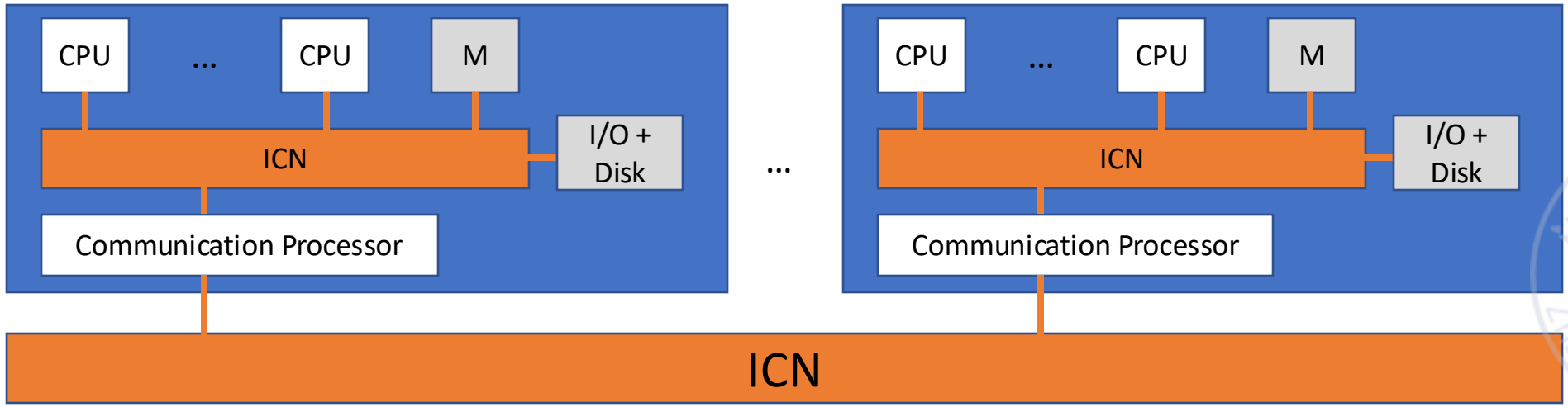
Multi-computer memory access model

- NORMA (No-Remote Memory Access)



Multi-computer system based on message passing

- General multi-computer architecture
  - Each node is composed of one or more CPUs, RAMs, disks and other input/output devices and communication processors.
  - The communication processors are connected to each other through an interconnection network. A variety of different topologies, switching strategies and path finding algorithms can be used.



MIMD Architecture

- Different memory access models of MIMD multi-processor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

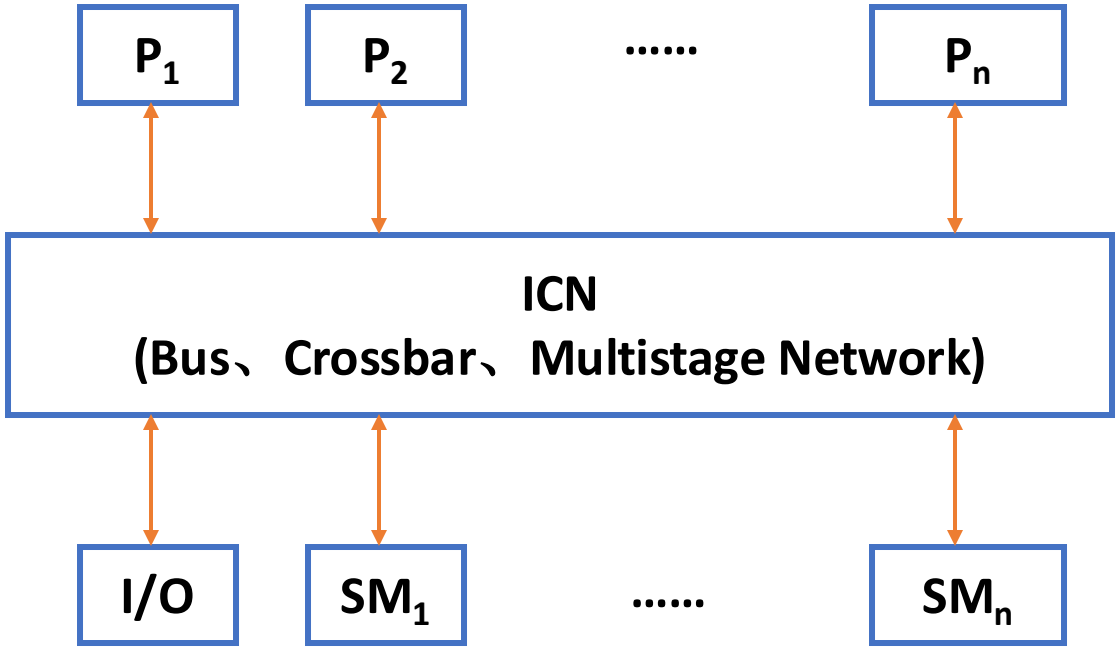
- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

MIMD Architecture

- Different memory access models of MIMD multiprocessor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

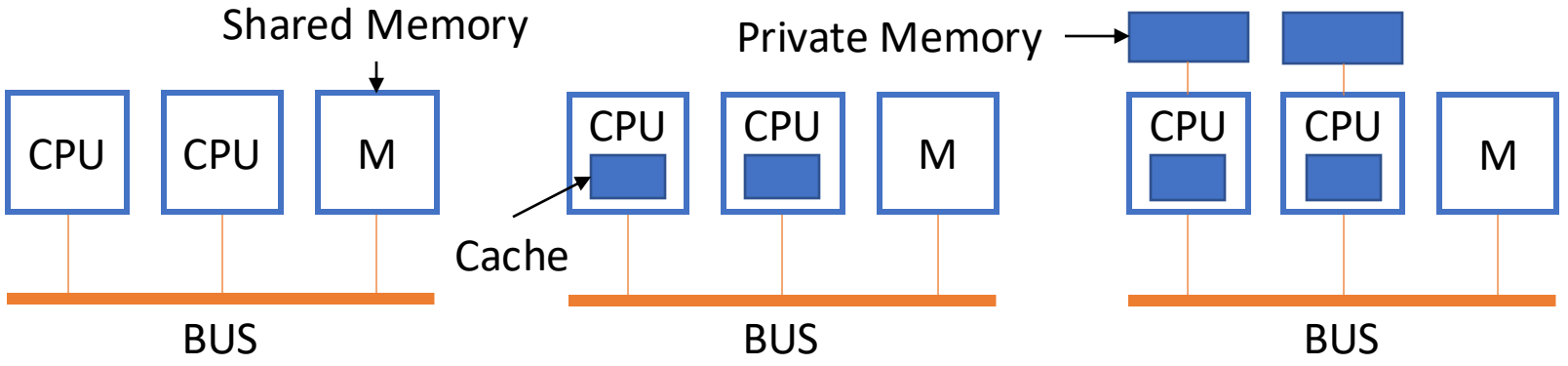
- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

UMA



UMA Multi-processor system

- UMA system features
  - Physical memory is uniformly shared by all processors
  - It takes the same time for all processors to access any memory word
  - Each processor can be equipped with private cache or private memory
- UMA Multiprocessor System Based on Bus

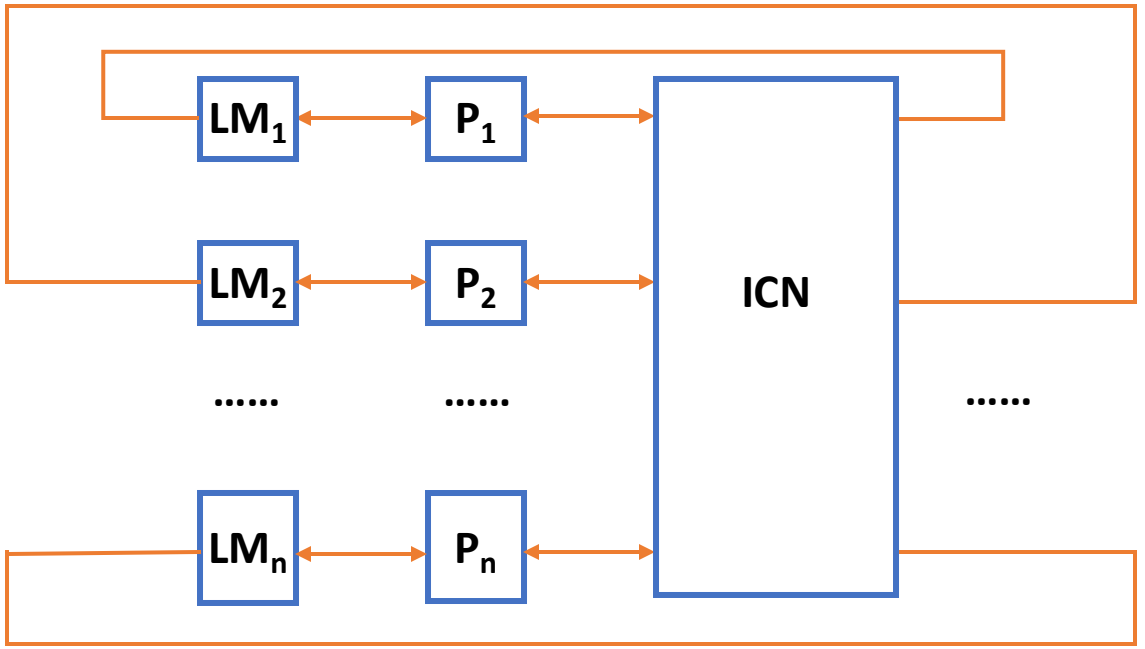


MIMD Architecture

- Different memory access models of MIMD multiprocessor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

NUMA

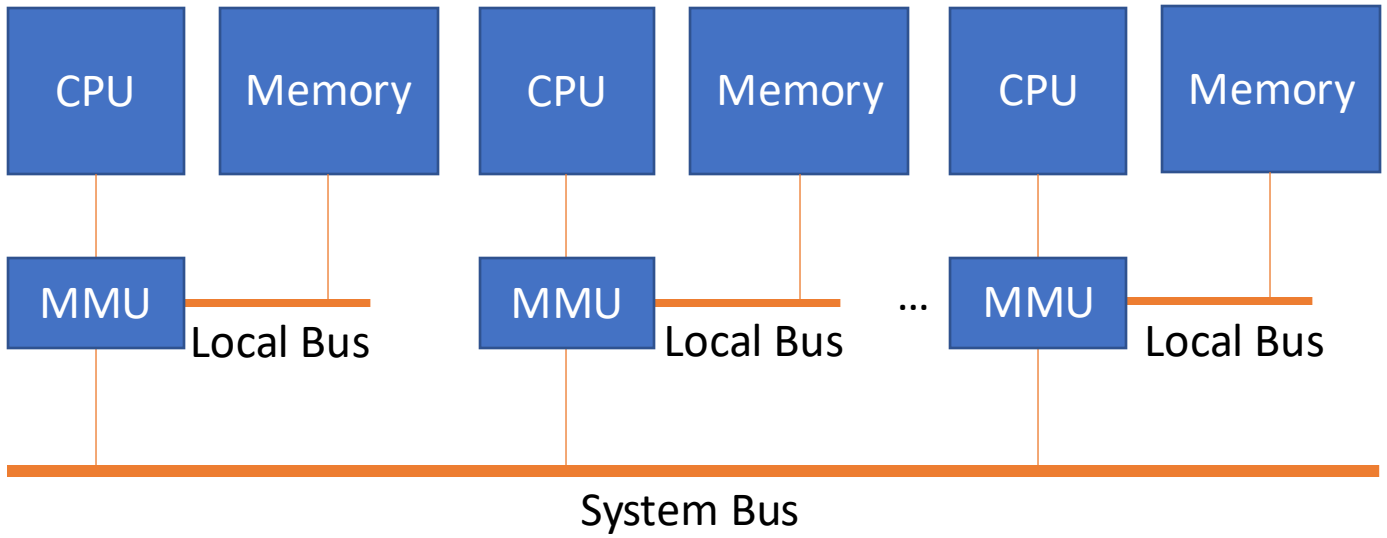


Shared local memory model LM: Local Memory P: Processor

NUMA

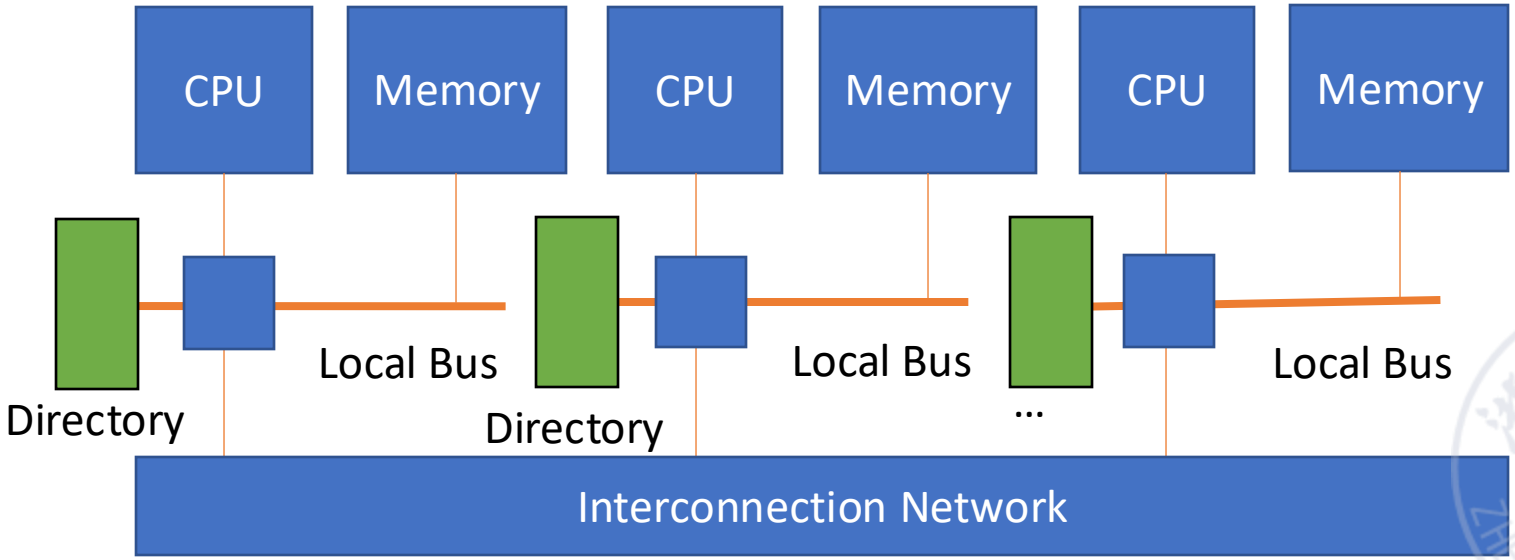
- NC-NUMA

  Non Cache
  Non-Uniform
  Memory Access



- CC-NUMA

  Coherent Cache
  Non-Uniform
  Memory Access



NUMA

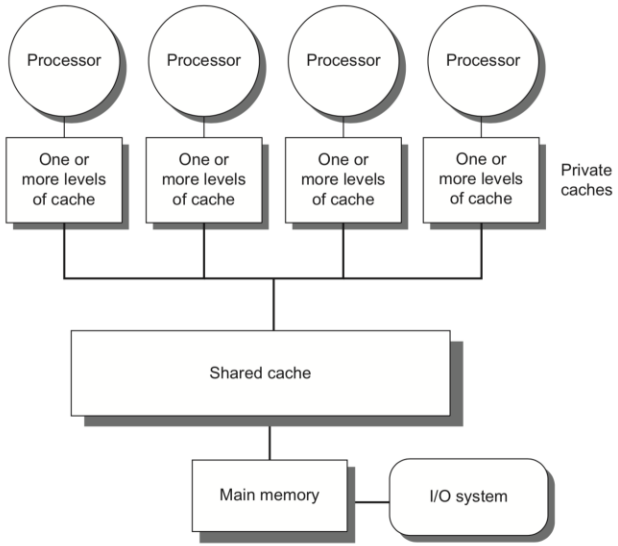
- NUMA system features
  - All CPUs share an uniform address space
  - Use LOAD and STORE instructions to access remote memory
  - Access to remote memory is slower than access to local memory
  - The processor in the NUMA system can use cache

- NC-NUMA and CC-NUMA
  - The NUMA system without Cache is called the NC-NUMA multiprocessor system, which means that the remote memory access time is not hidden in this system. If Cache is used, the system will be called a CC-NUMA multiprocessor system.

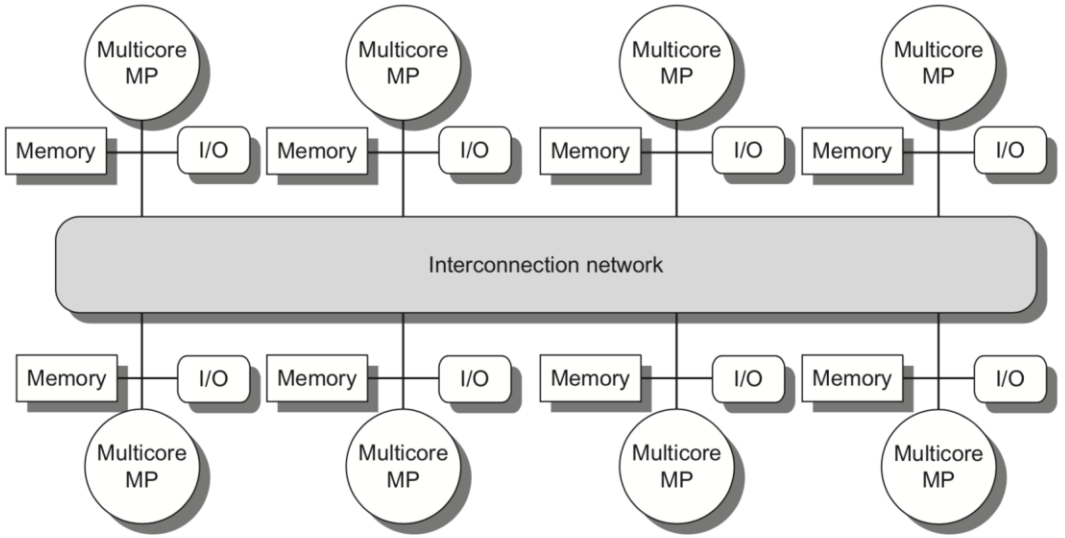
UMA and NUMA

- UMA is also called *symmetric (shared-memory) multiprocessors (SMP)* or *centralized shared-memory multiprocessors*.
- NUMA is called *distributed shared-memory multiprocessor (DSP)*.

SMP



DSP

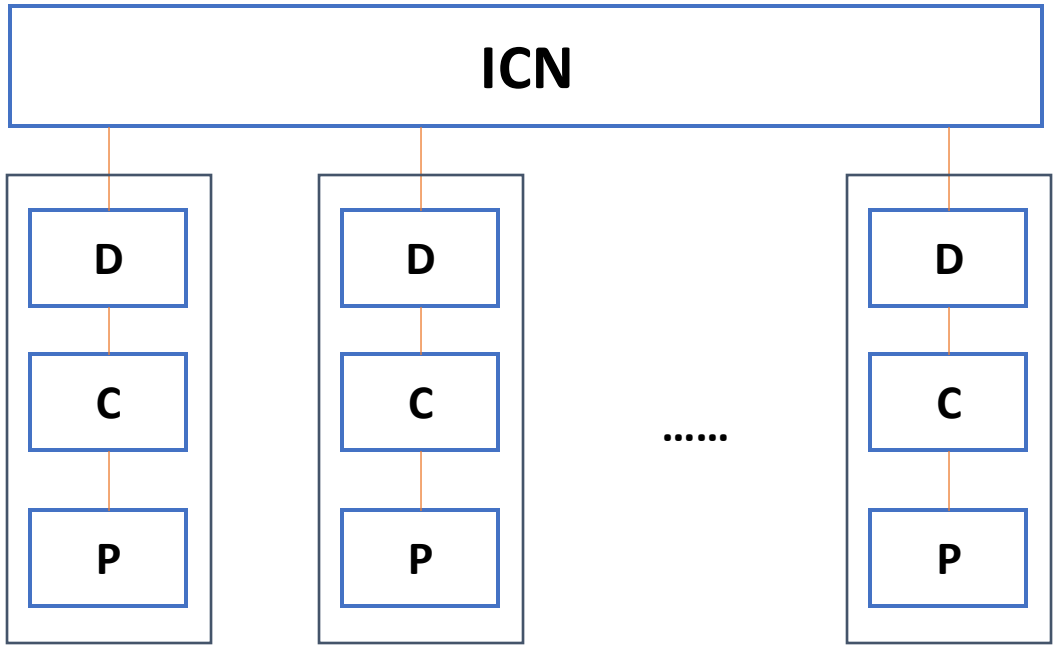


MIMD Architecture

- Different memory access models of MIMD multiprocessor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

COMA

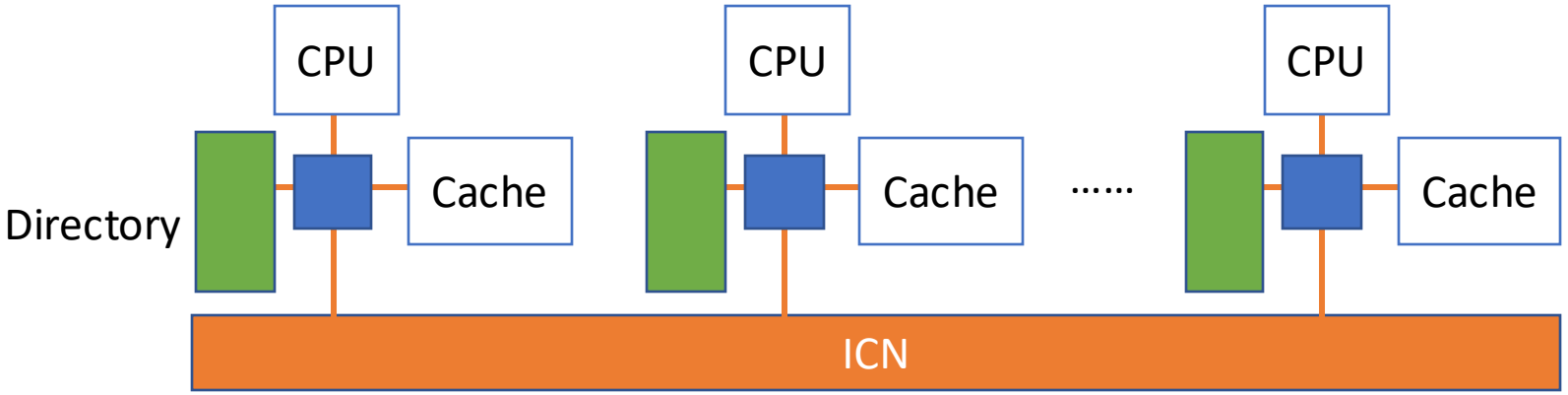


D: Cache Directory **C:** Cache **P:** Processor

COMA

Characteristics of COMA

- COMA is a special case of NUMA. There is no storage hierarchy in each processor node, and all caches form a uniform address space.
- Use the distributed cache directory for remote cache access. When using COMA, the data can be allocated arbitrarily at the beginning, because it will eventually be moved to the place where it is used at runtime.



Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task.

The first hurdle has to do with the limited parallelism available in programs.

The second arises from the relatively high cost of communications.

Both can be explained with Amdahl's Law.

Challenges of Parallel Processing

Example：

Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of the original computation can be sequential!

Challenges of Parallel Processing

Example:

Suppose we have an application running on a 100-processor multiprocessor, and assume that application can use 1, 50, or 100 processors. If we assume that 95% of the time we can use all 100 processors, how much of the remaining 5% of the execution time must employ 50 processors if we want a speedup of 80?

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{100}}{\text{Speedup}_{100}} + \frac{\text{Fraction}_{50}}{\text{Speedup}_{50}} + (1 - \text{Fraction}_{100} - \text{Fraction}_{50})}$$

$$80 = \frac{1}{\frac{0.95}{100} + \frac{\text{Fraction}_{50}}{50} + (1 - 0.95 - \text{Fraction}_{80})}$$

$$0.76 + 1.6 \times \text{Fraction}_{50} + 4.0 - 80 \times \text{Fraction}_{50} = 1$$

$$4.76 - 78.4 \times \text{Fraction}_{50} = 1$$

$$\text{Fraction}_{50} = 0.048$$

If 95% of an application can use 100 processors perfectly, to get a speedup of 80, 4.8% of the remaining time must be spent using 50 processors and only 0.2% can be serial!

Challenges of Parallel Processing

Example：

Suppose we have an application running on a 32-processor multiprocessor that has a 100 ns delay to handle a reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy. Processors are stalled on a remote request, and the processor clock rate is 4 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, and 0.2% of the instructions involve a remote communication reference, how much faster is the multiprocessor with no communication?

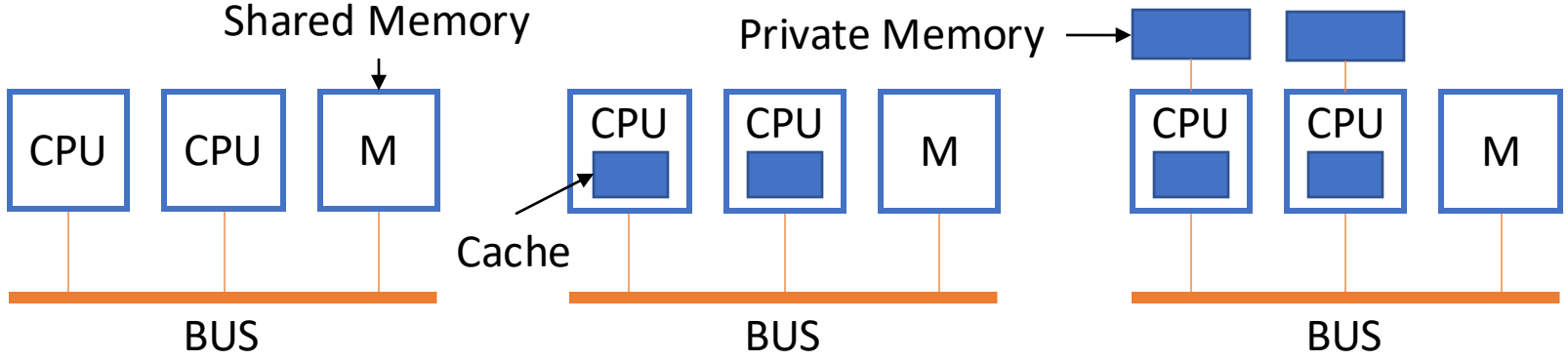
$$CPI = Base\ CPI + Remote\ request\ rate \times Remote\ request\ cost = 0.5 + 0.2\% \times Remote\ request\ cost$$

$$The\ remote\ request\ cost\ is:\quad Remote\ access\ cost / Cycle\ time = 100ns / 0.25ns = 400\ cycles$$

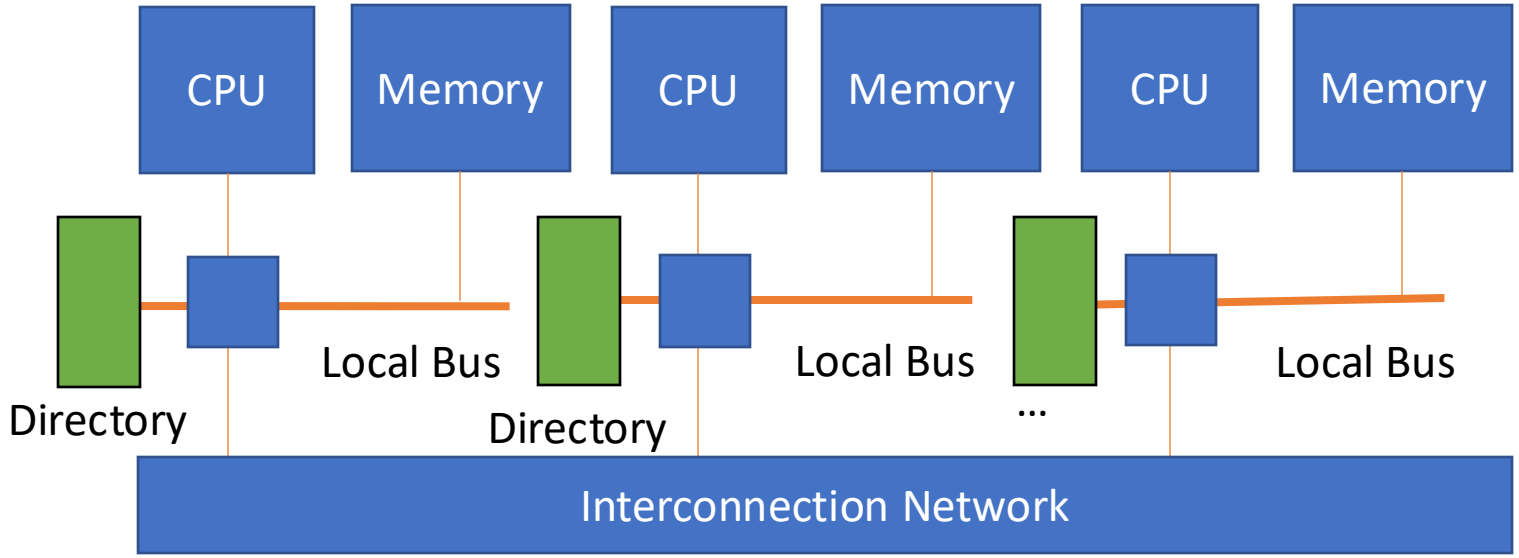
$$SP = CPI / Base\ CPI = (0.5 + 0.20\% \times 400) / 0.5 = 1.3 / 0.5 = 2.6$$

Challenges of Multi-process system based on Shared Memory

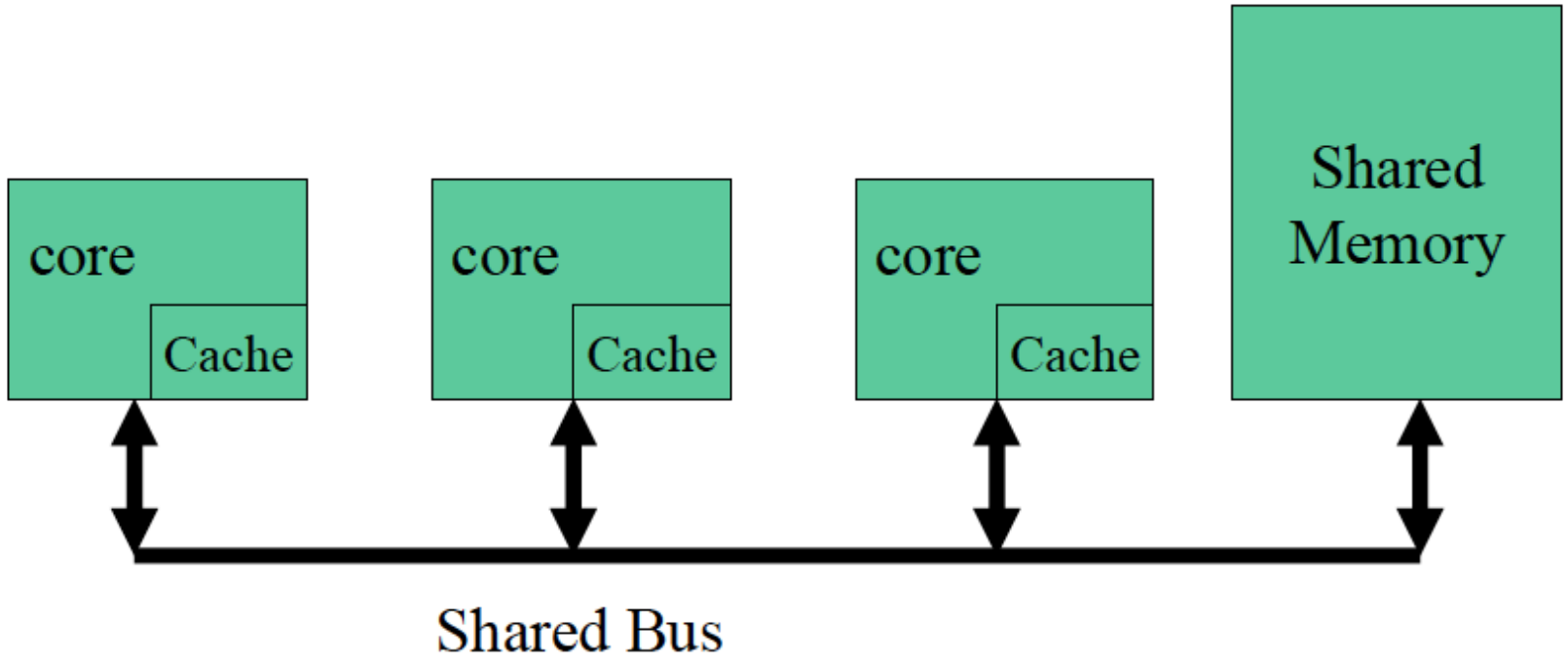
UMA



NUMA



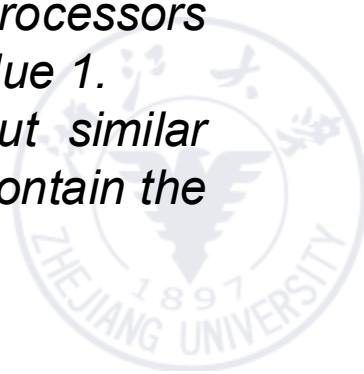
Challenges of Multi-process system based on Shared Memory



Challenges of Multi-process system based on Shared Memory

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a writethrough cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!



Memory Consistency and Cache Coherence

- Memory Consistency Need Memory Consistency Model
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- Cache Coherence Need Cache Coherence Protocol
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors
  - Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. This is because correct coherence ensures that the caches never enable new or different *functional* behavior.

Cache Coherence

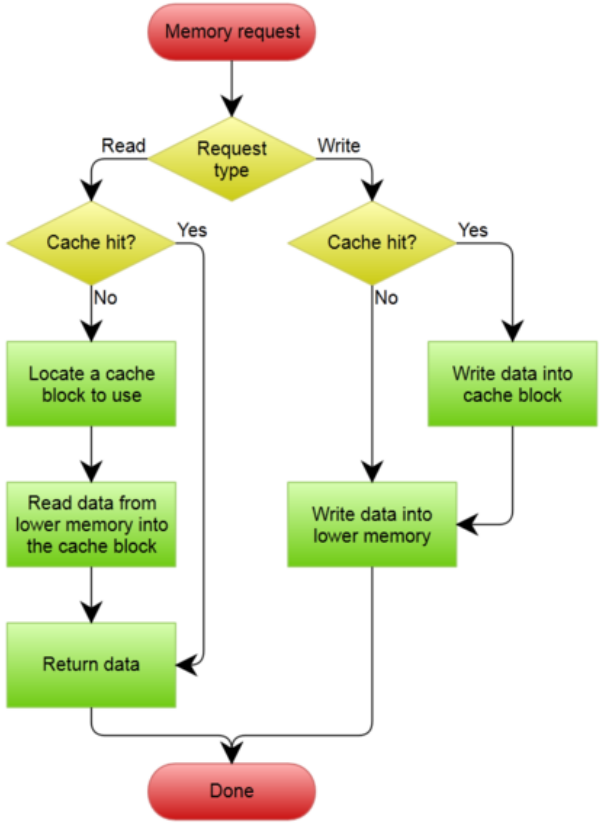
- Causes of Cache coherence problems
  - In modern parallel computers, processors often have Cache. One memory data may have multiple copies in the entire system. This leads to the Cache coherence problem.
- Cache coherence protocol
  - A set of rules implemented by Cache, CPU, and memory to prevent different versions of the same data from appearing in multiple Caches forms a Cache coherence protocol.
- Cache coherence protocols can generally be divided into two categories
  - Bus snooping protocol
  - Directory based protocol

Cache Coherence

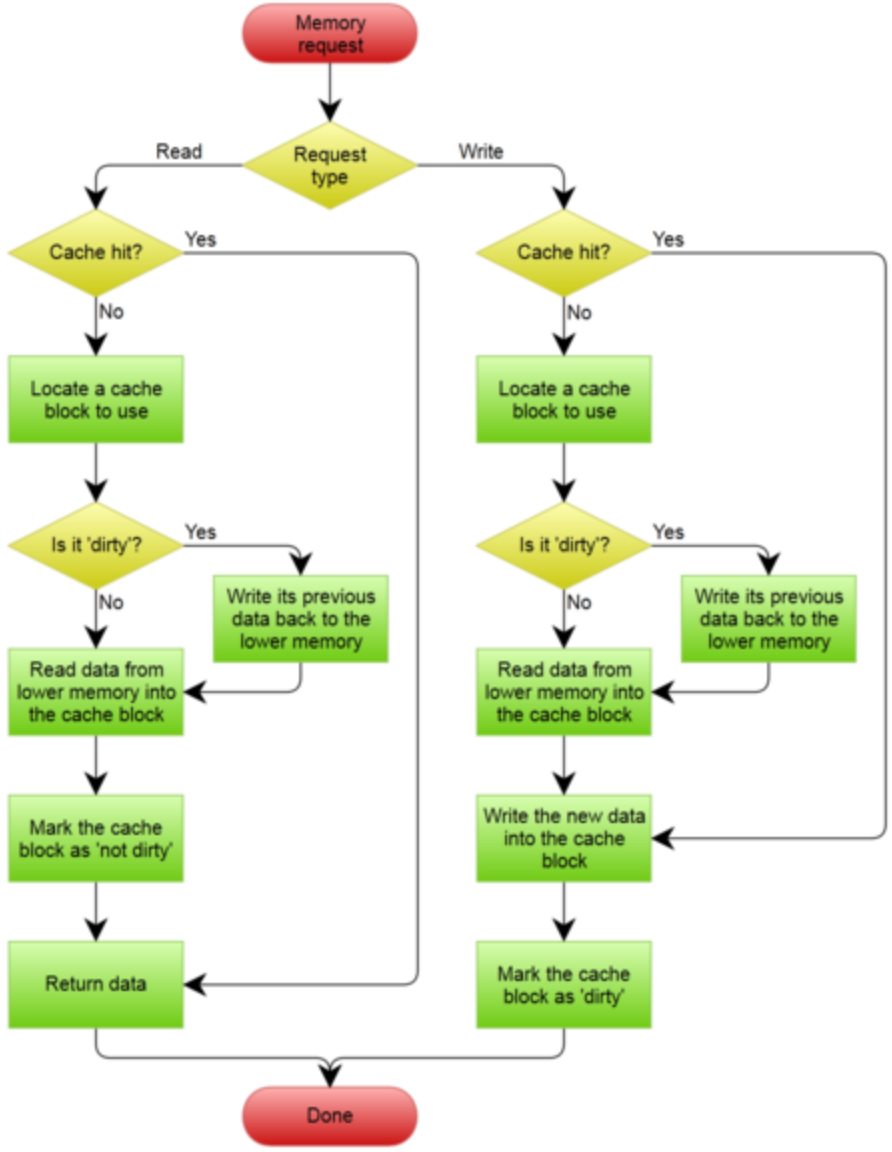
- For UMA: *Snoopy coherence protocols*
  - In the snoopy coherence protocols, all processors snoop the bus. When a processor modifies the data in the private cache, it broadcasts invalid information or updated data on the bus to invalidate or update other copies.

- For NUMA: *Directory protocol*
  - The directory protocol uses a directory to record which processors in the system have copies of certain storage blocks in the cache. When a processor wants to write a shared block, it sends an invalid signal to those processors that have copies of the block through the directory in a "point-to-point" way, so that all other copies are invalidated.

Cache Coherence



A Write-Through cache with No-Write Allocation



A Write-Back cache with Write Allocation

Snoopy Coherence Protocols

- Write-through
  - While writing the data in the cache line, the content in the corresponding memory is also modified, and the data in the memory is kept up to date at any time.

- Write-back
  - The write operation does not directly write to the memory. On the contrary, when the cache line is modified, a certain bit in the cache is set to indicate that the data in the cache line is correct but the data in the memory is out of date. Of course, the line will eventually be written back to memory, but it may be after multiple write operations.

Write-through Cache Coherency Protocol

- Four situations when the monitoring cache performs read and write operations according to this protocol

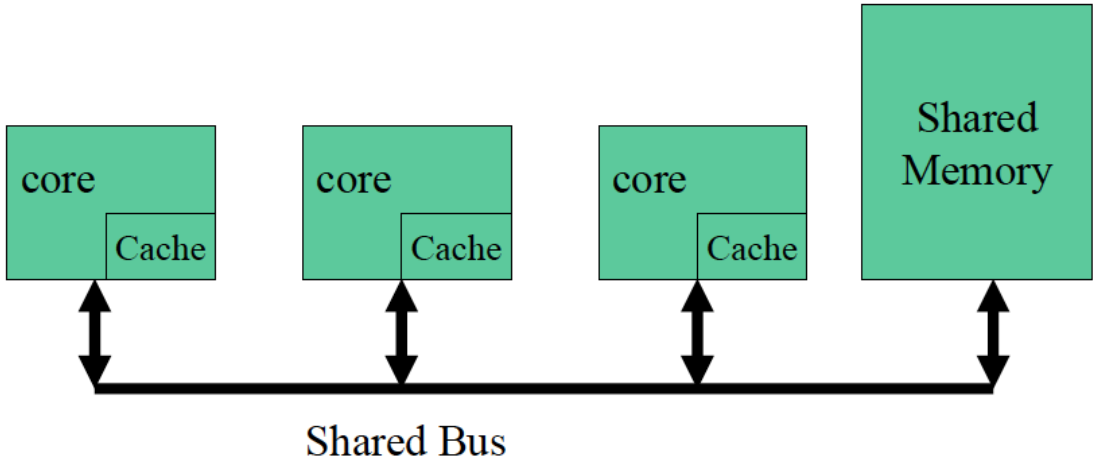
	Local Request	Remote Request
Read Miss	Access data from memory	
Read Hit	Use local cache data	
Write Miss	Modify data in memory	
Write Hit	Modify cache and memory	Invalidate the cache item

- There are many changes in the basic protocol
  - Whether to use **Update Strategy** or **Invalidate Strategy** for remote write hits
  - Whether to transfer the corresponding word into the cache when the cache write is missing, this is whether to use the **Write-allocate Policy**.

Memory Consistency and Cache Coherence

- Memory Consistency Need Memory Consistency Model
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- Cache Coherence Need Cache Coherence Protocol
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors
  - Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. This is because correct coherence ensures that the caches never enable new or different *functional* behavior.

Cache Coherence



- For UMA: *Snoopy coherence protocols*
  - In the snoopy coherence protocols, all processors snoop the bus. When a processor modifies the data in the private cache, it broadcasts invalid information or updated data on the bus to invalidate or update other copies.

- For NUMA: *Directory protocol*
  - The directory protocol uses a directory to record which processors in the system have copies of certain storage blocks in the cache. When a processor wants to write a shared block, it sends an invalid signal to those processors that have copies of the block through the directory in a "point-to-point" way, so that all other copies are invalidated.

Snoopy Coherence Protocols

- Write invalidate protocol

  It invalidates other copies on a write.

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

An invalidation protocol works on a snooping bus for a single cache block (X) with write-back caches.

Snoopy Coherence Protocols

- Write invalidate protocol

  It invalidates other copies on a write.

- Write update / write broadcast protocol

  Update all the cached copies of a data item when that item is written.

Snoopy Coherence Protocols

- Write invalidate protocol

  Implementation

  three block states ***(MSI protocol)***

  - ***I*nvalid**
  - ***S*hared**

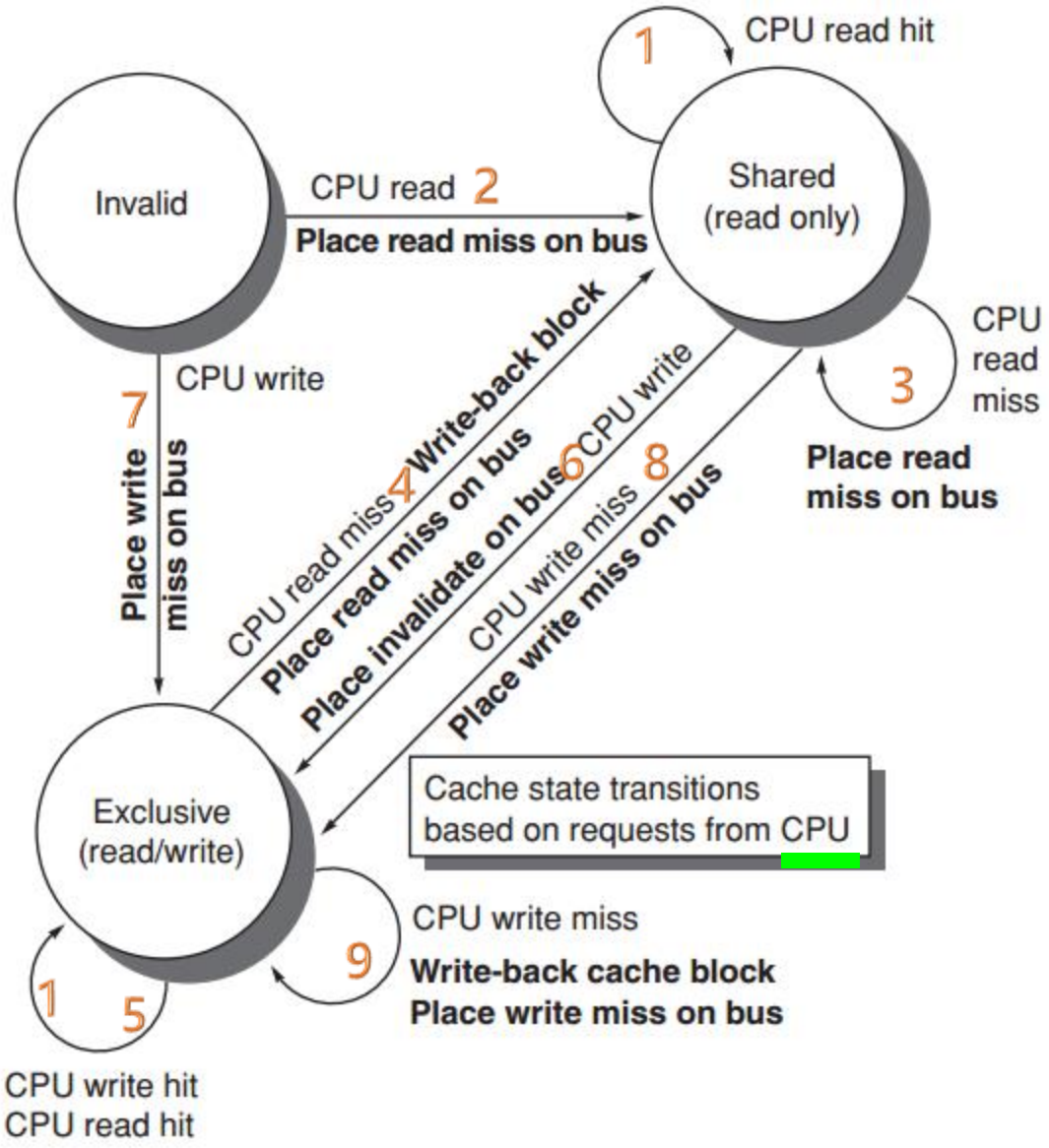
    indicates that the block in the private cache is potentially shared
  - ***M*odified**

    indicates that the block has been updated in the private cache;

    implies that the block is **exclusive**

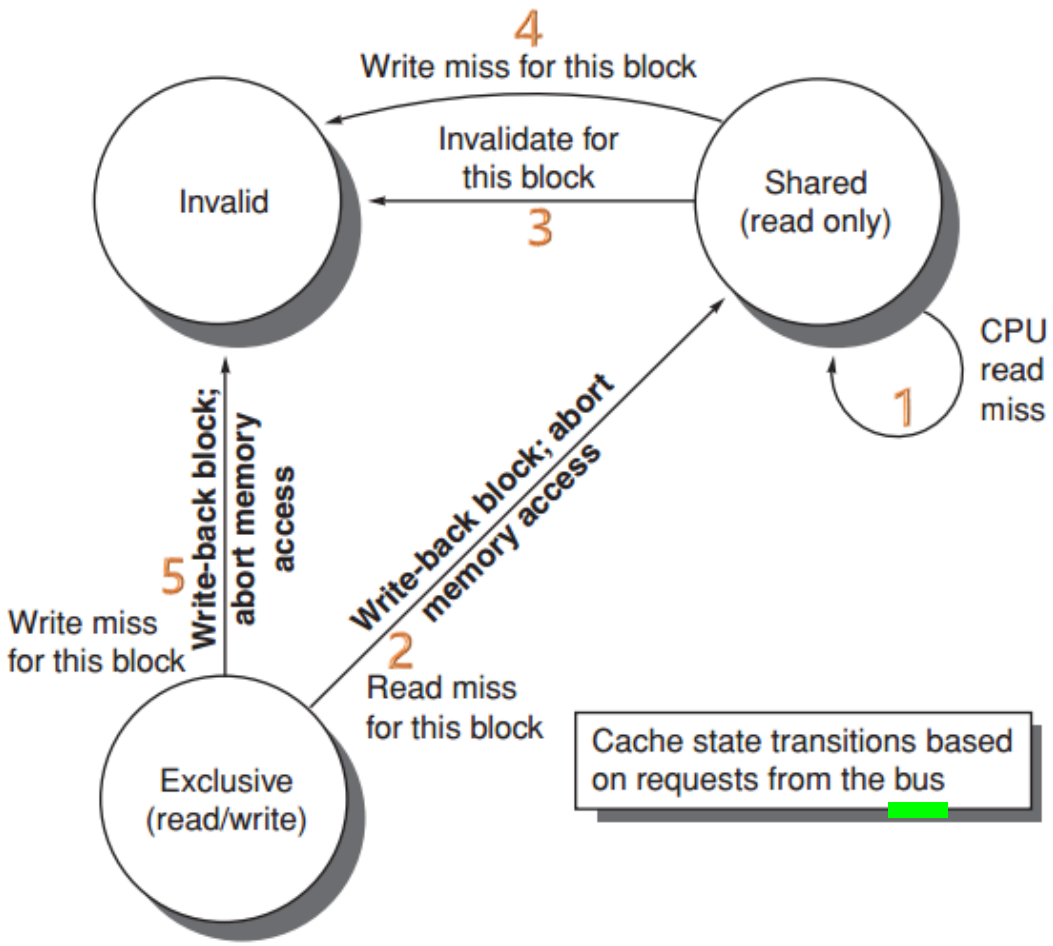
Write Invalidation Protocol （write back）

Request	Source	State of addressed cache block	Type of cache action	Function and explanation	
Read hit	processor	shared or modified	normal hit	Read data in cache.	1
Read miss	processor	invalid	normal miss	Place read miss on bus.	2
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.	3
Read miss	processor	modified	replacement	Address conflict miss: write back block, then place read miss on bus.	4
Write hit	processor	modified	normal hit	Write data in cache.	5
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.	6
Write miss	processor	invalid	normal miss	Place write miss on bus.	7
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.	8
Write miss	processor	modified	replacement	Address conflict miss: write back block, then place write miss on bus.	9
Read miss	bus	shared	no action	Allow memory to service read miss.	
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.	
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.	
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.	
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid.	

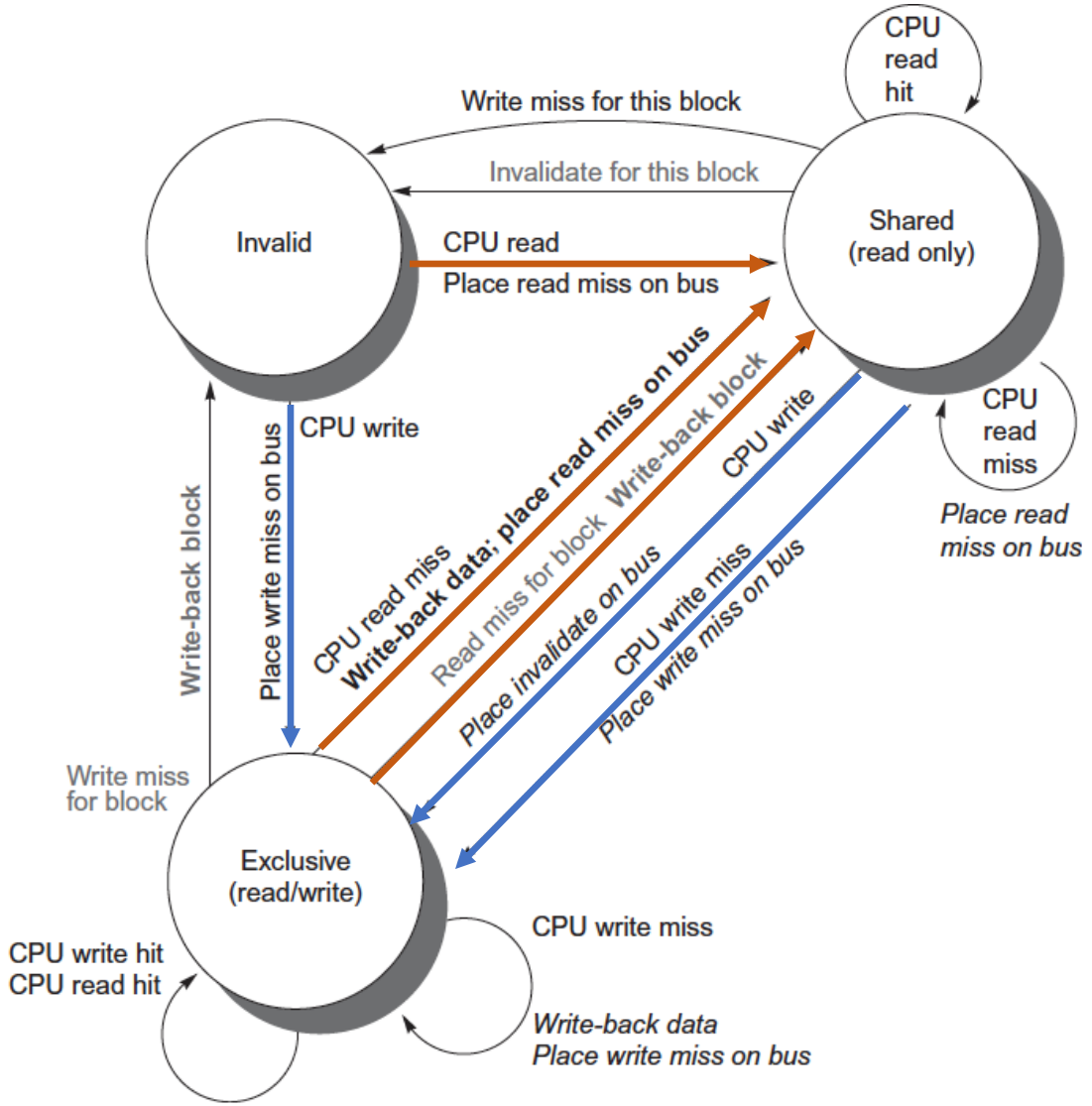


Write Invalidation Protocol （write back）

Request	Source	State of addressed cache block	Type of cache action	Function and explanation	
Read hit	processor	shared or modified	normal hit	Read data in cache.	
Read miss	processor	invalid	normal miss	Place read miss on bus.	
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.	
Read miss	processor	modified	replacement	Address conflict miss: write back block, then place read miss on bus.	
Write hit	processor	modified	normal hit	Write data in cache.	
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.	
Write miss	processor	invalid	normal miss	Place write miss on bus.	
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.	
Write miss	processor	modified	replacement	Address conflict miss: write back block, then place write miss on bus.	
Read miss	bus	shared	no action	Allow memory to service read miss.	1
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.	2
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.	3
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.	4
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid.	5



Write Invalidation Protocol （write back）



MSI protocol

The initial cache and memory state for a shared memory multi-processor system are shown in the following figure. Each core processor has a **4-line direct-mapped write back** cache. Basic **write invalidation snooping protocol** is used. Please show the action results (ie. updated parts in the caches and memory) for the three action as that in the example, assuming that all the actions are applied to the initial cache and memory states in the figure. In cache state, I means Invalid, S means Shared, M means Modified.

Action specification format:

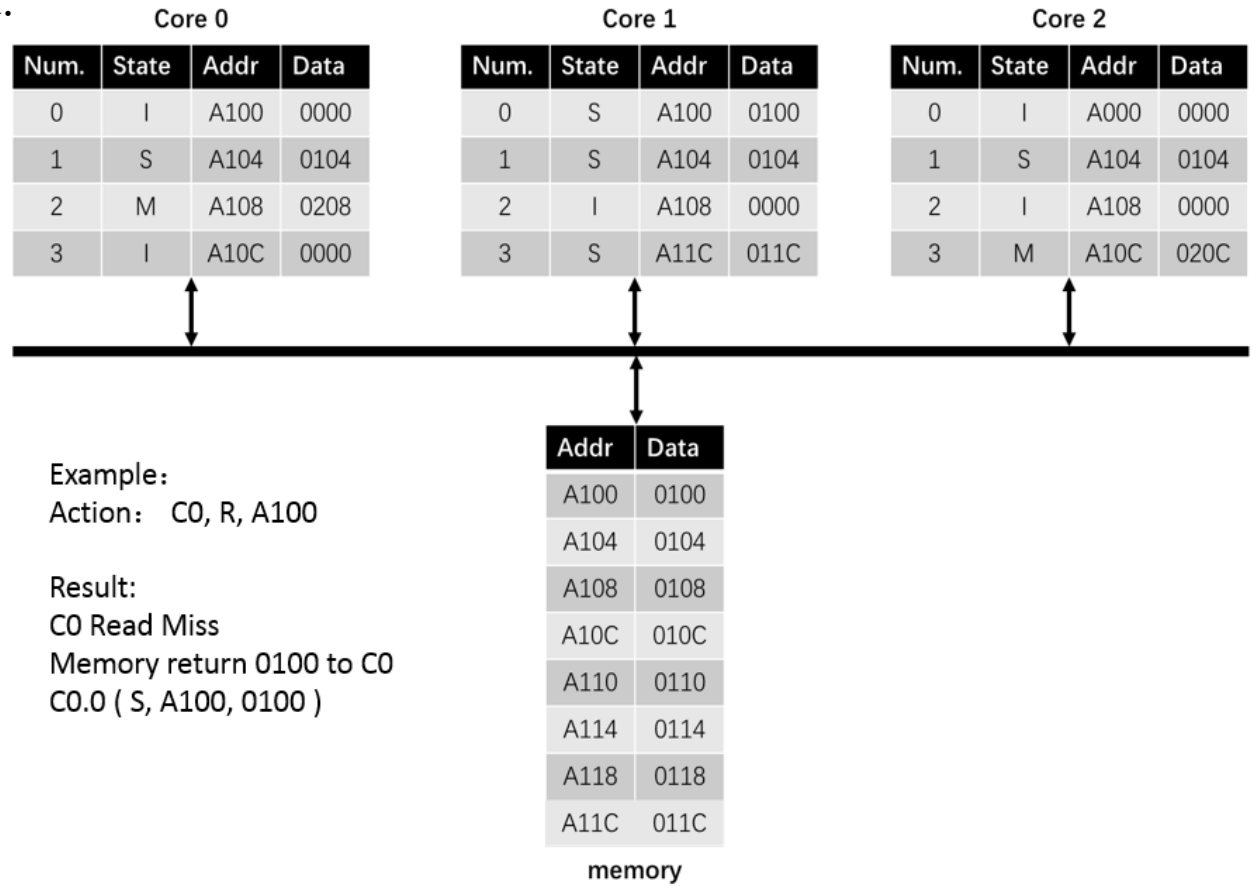
C#, R, <address Axxx> represents
Core # reads address Axxx.

C#, W, <address Axxx>, <value V> represents
Core # writes value V into address Axxx.

Cx.y represents cache line y in core x.

Memory A100, 0000 → 0100 represents
the value in memory location A100 is updated from 0000 to 0100.

(1) Action: C0, R, A10C
(2) Action: C1, W, A104, 0204
(3) Action: C0, W, A118, 0308



MSI protocol

The initial cache and memory state for a shared memory multi-processor system are shown in the following figure. Each core processor has a **4-line direct-mapped write back** cache. Basic **write invalidation snooping protocol** is used. Please show the action results (ie. updated parts in the caches and memory) for the three action as that in the example, assuming that all the actions are applied to the initial cache and memory states in the figure. In cache state, I means Invalid, S means Shared, M means Modified.

Action specification format:

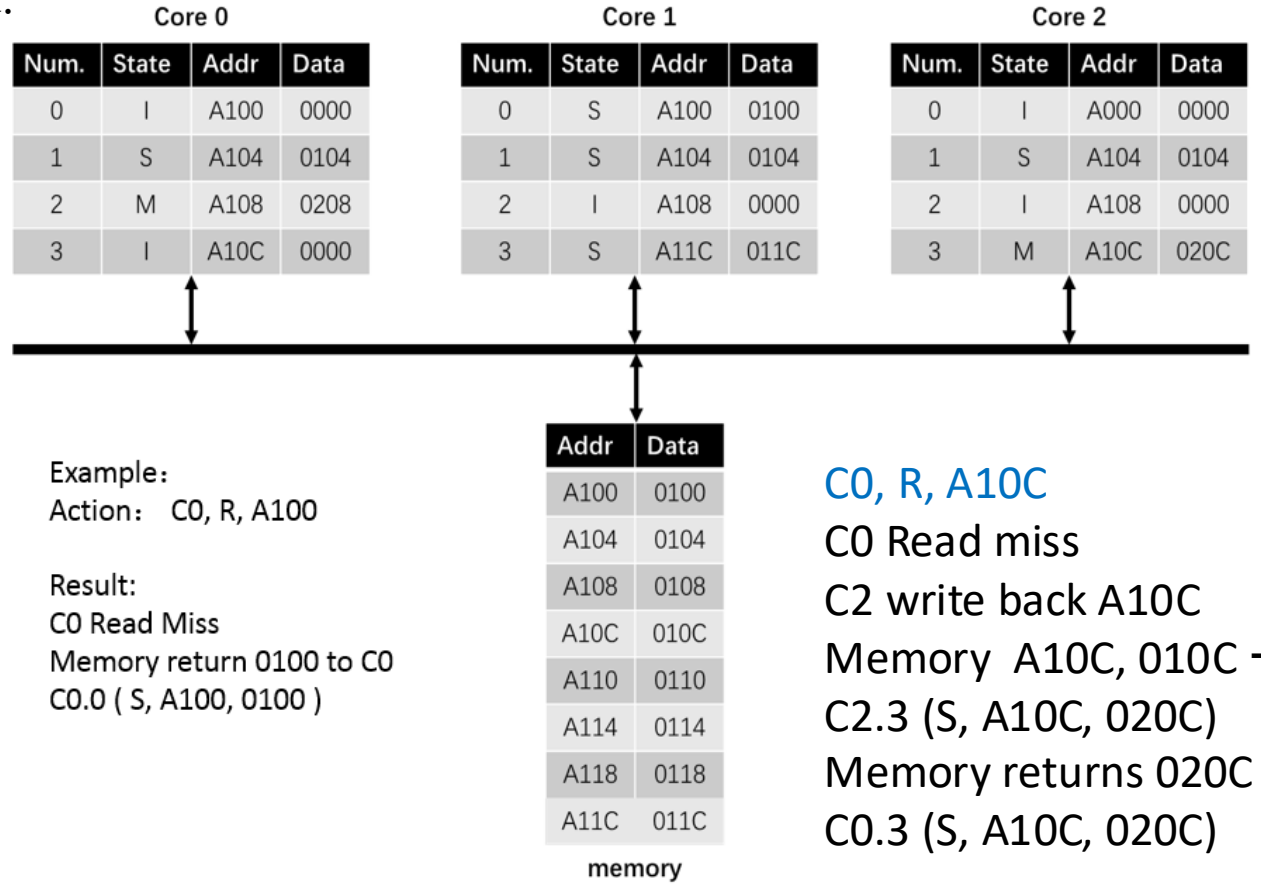
C#, R, <address Axxx> represents
Core # reads address Axxx.

C#, W, <address Axxx>, <value V> represents
Core # writes value V into address Axxx.

Cx.y represents cache line y in core x.

Memory A100, 0000 → 0100 represents
the value in memory location A100 is updated from 0000 to 0100.

(1) Action: C0, R, A10C
(2) Action: C1, W, A104, 0204
(3) Action: C0, W, A118, 0308



Example:
Action: C0, R, A100

Result:
C0 Read Miss
Memory return 0100 to C0
C0.0 (S, A100, 0100)

C0, R, A10C
C0 Read miss
C2 write back A10C
Memory A10C, 010C → 020C
C2.3 (S, A10C, 020C)
Memory returns 020C to C0
C0.3 (S, A10C, 020C)

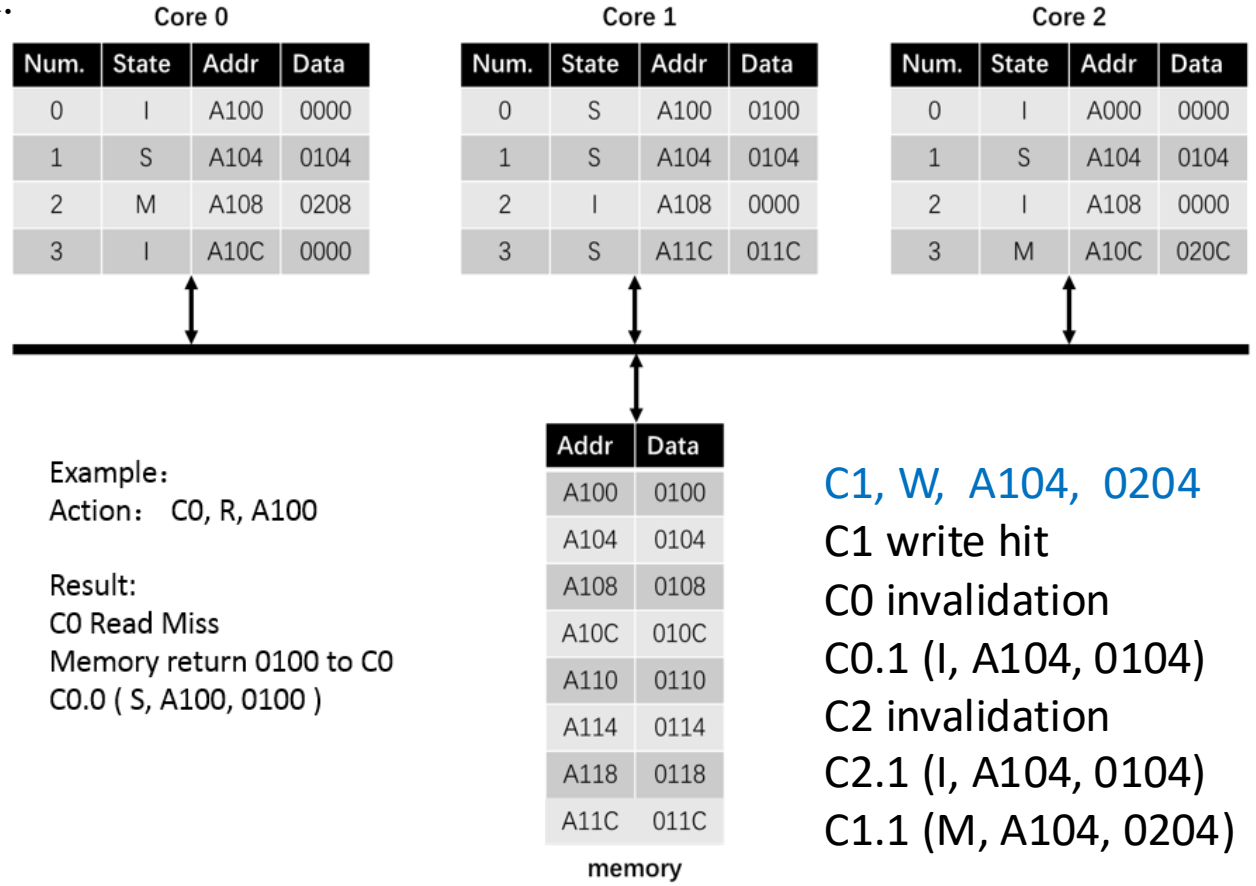
MSI protocol

The initial cache and memory state for a shared memory multi-processor system are shown in the following figure. Each core processor has a **4-line direct-mapped write back** cache. Basic **write invalidation snooping protocol** is used. Please show the action results (ie. updated parts in the caches and memory) for the three action as that in the example, assuming that all the actions are applied to the initial cache and memory states in the figure. In cache state, I means Invalid, S means Shared, M means Modified.

Action specification format:

C#, R, <address Axxx> represents
Core # reads address Axxx.
C#, W, <address Axxx>, <value V> represents
Core # writes value V into address Axxx.
Cx.y represents cache line y in core x.
Memory A100, 0000 → 0100 represents
the value in memory location A100 is updated from 0000 to 0100.

(1) Action: C0, R, A10C
(2) Action: C1, W, A104, 0204
(3) Action: C0, W, A118, 0308



MSI protocol

The initial cache and memory state for a shared memory multi-processor system are shown in the following figure. Each core processor has a **4-line direct-mapped write back** cache. Basic **write invalidation snooping protocol** is used. Please show the action results (ie. updated parts in the caches and memory) for the three action as that in the example, assuming that all the actions are applied to the initial cache and memory states in the figure. In cache state, I means Invalid, S means Shared, M means Modified.

Action specification format:

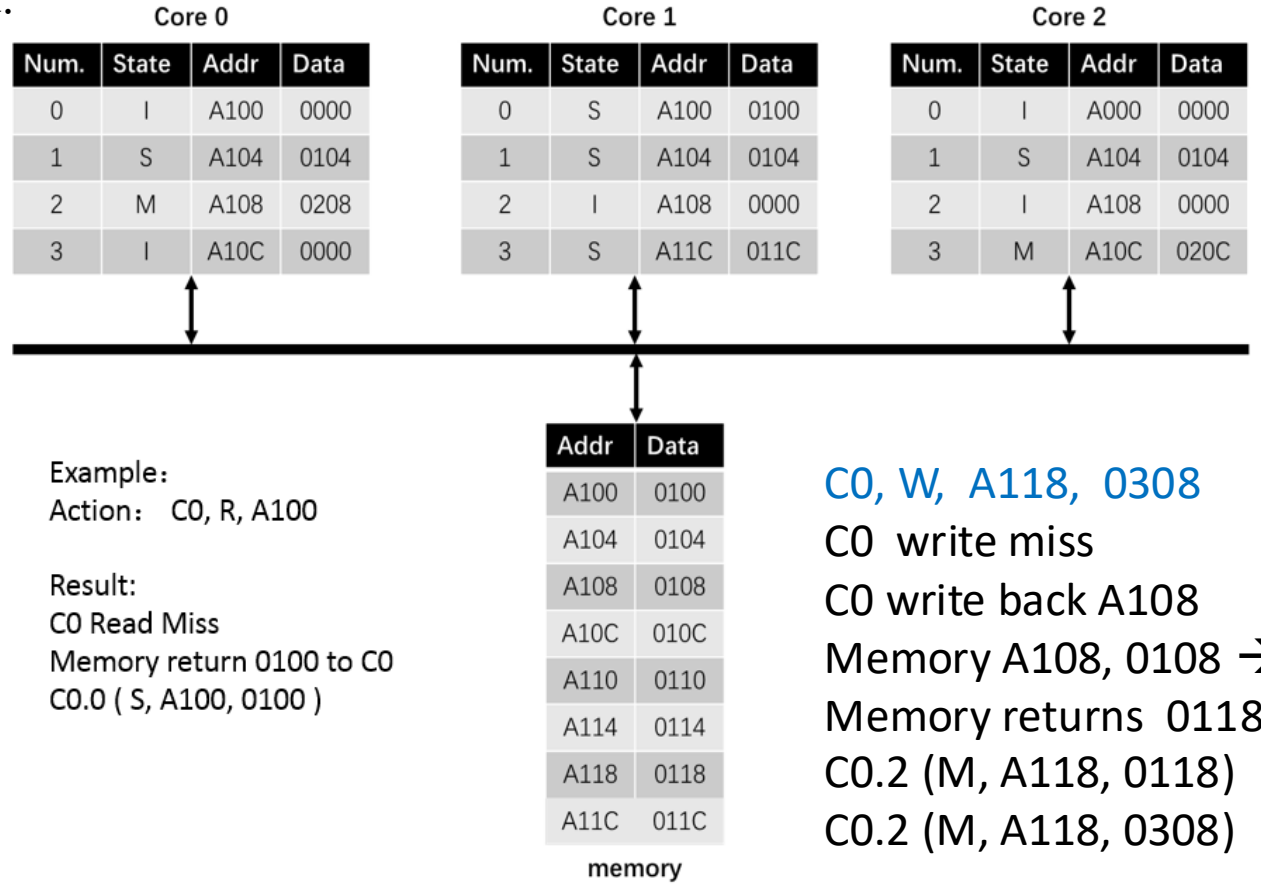
C#, R, <address Axxx> represents
Core # reads address Axxx.

C#, W, <address Axxx>, <value V> represents
Core # writes value V into address Axxx.

Cx.y represents cache line y in core x.

Memory A100, 0000 → 0100 represents
the value in memory location A100 is updated from 0000 to 0100.

(1) Action: C0, R, A10C
(2) Action: C1, W, A104, 0204
(3) Action: C0, W, A118, 0308



Example:
Action: C0, R, A100

Result:
C0 Read Miss
Memory return 0100 to C0
C0.0 (S, A100, 0100)

C0, W, A118, 0308
C0 write miss
C0 write back A108
Memory A108, 0108 → 0208
Memory returns 0118 to C0
C0.2 (M, A118, 0118)
C0.2 (M, A118, 0308)

MSI Protocol Extensions

- **MESI**

  exclusive: indicates when a cache block is resident only in a single cache but is clean

  exclusive-> read by others / read miss ->shared

  exclusive-> write ->modified

  MESI writes exclusive to modified silently, without broadcast on bus

MSI Protocol Extensions

- **MOESI**

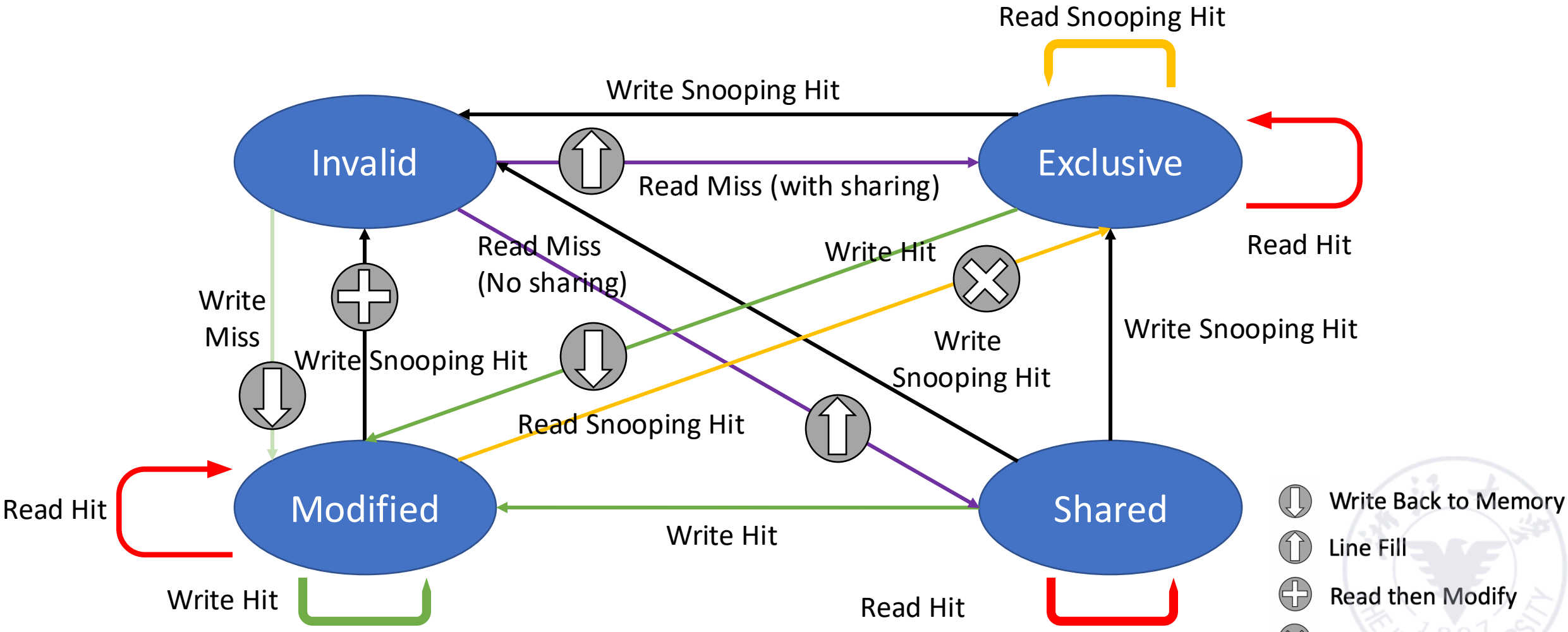
  owned: indicates that the associated block is owned by that cache and out-of-date in memory

  Modified -> Owned without writing the shared block to memory

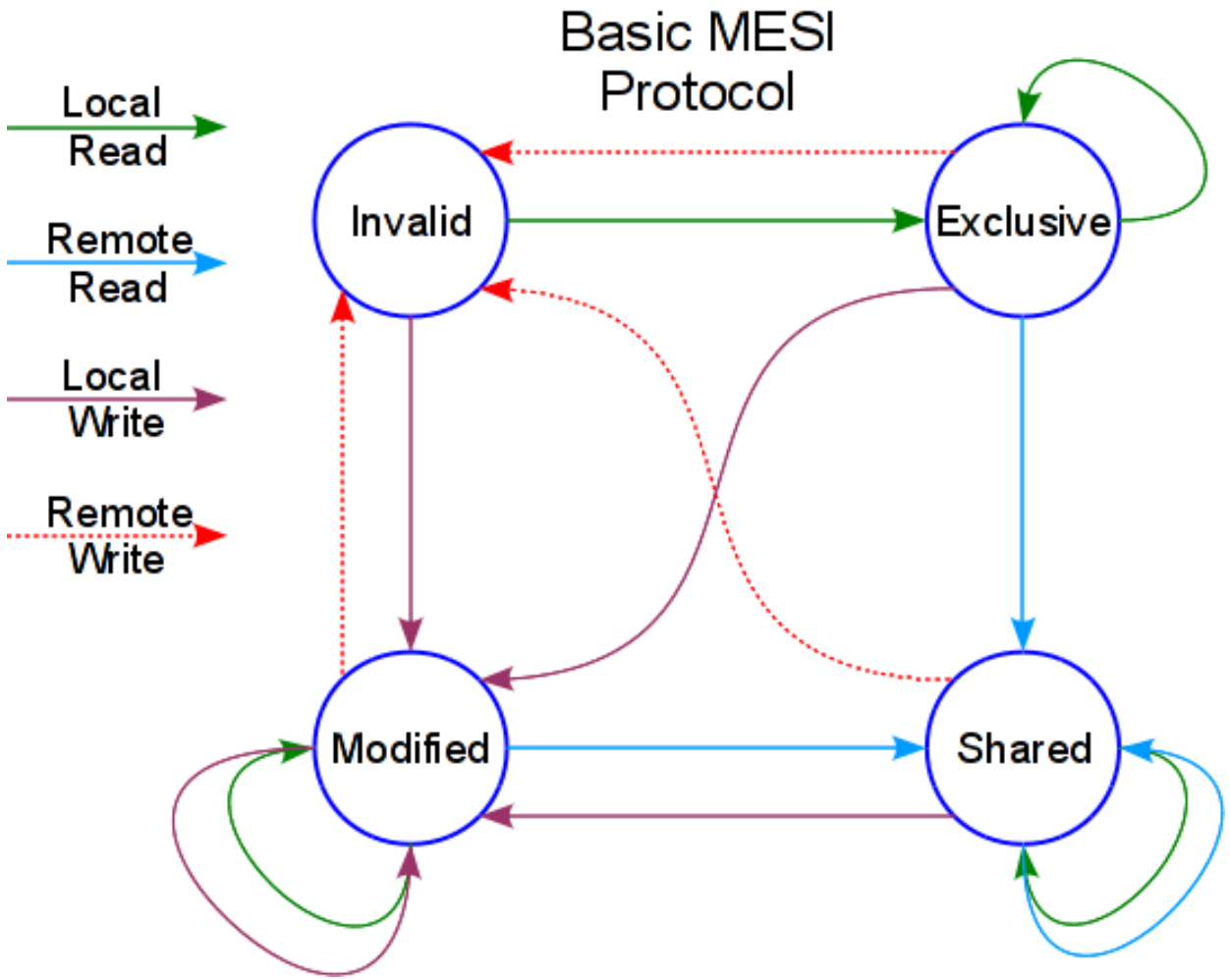
MSI Protocol Extensions (write back)

- MESI protocol: It is named after the initial letter of the four states used in the protocol. Each item in this protocol is in one of the following four states:
  - Invalid: The data contained in the cache item is invalid.
  - Shared: This row of data exists in multiple cache items, and the data in the memory is the latest.
  - Exclusive: No other cache items include this row of data, and the data in memory is the latest.
  - Modified: The data of the item is valid, but the data in the memory is invalid, and there is no copy of the data in other cache items.

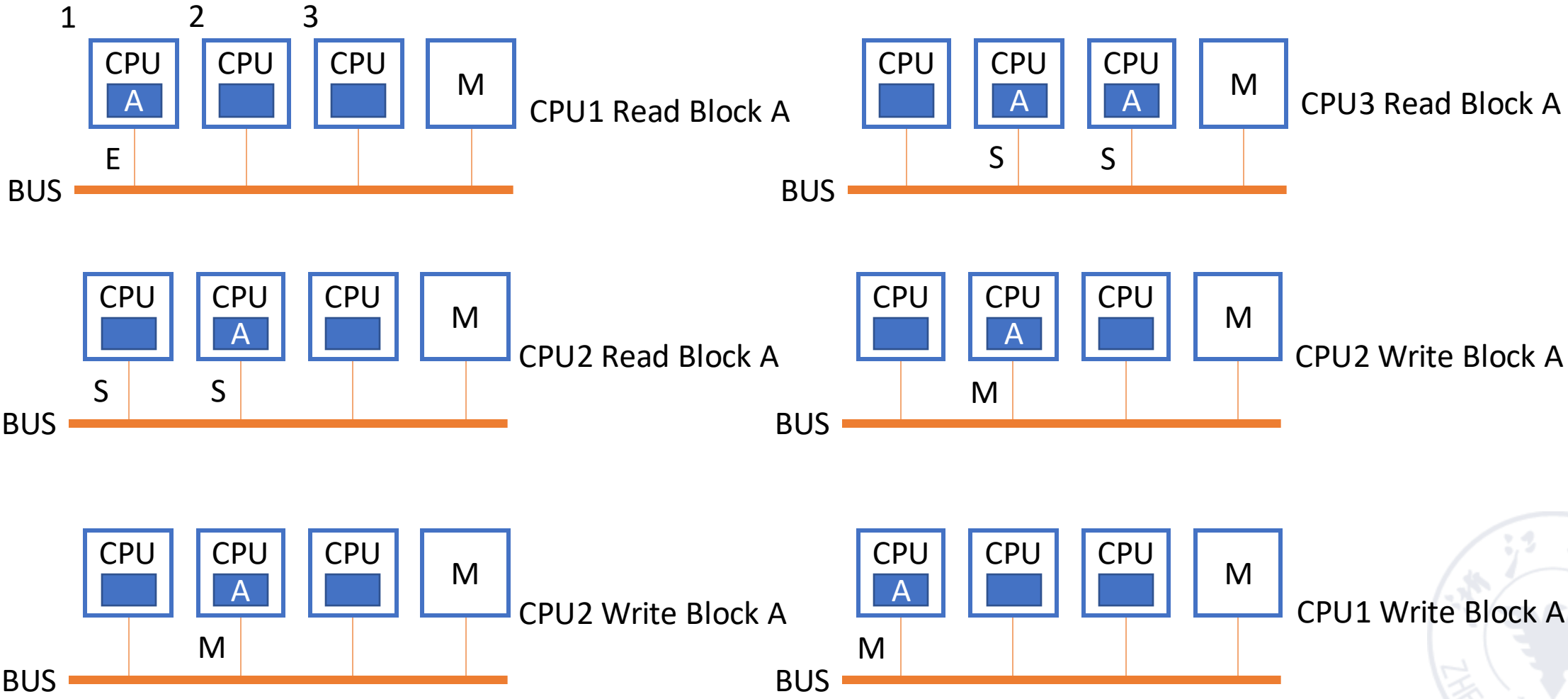
MESI Protocol (write back)



MESI Protocol (write back)

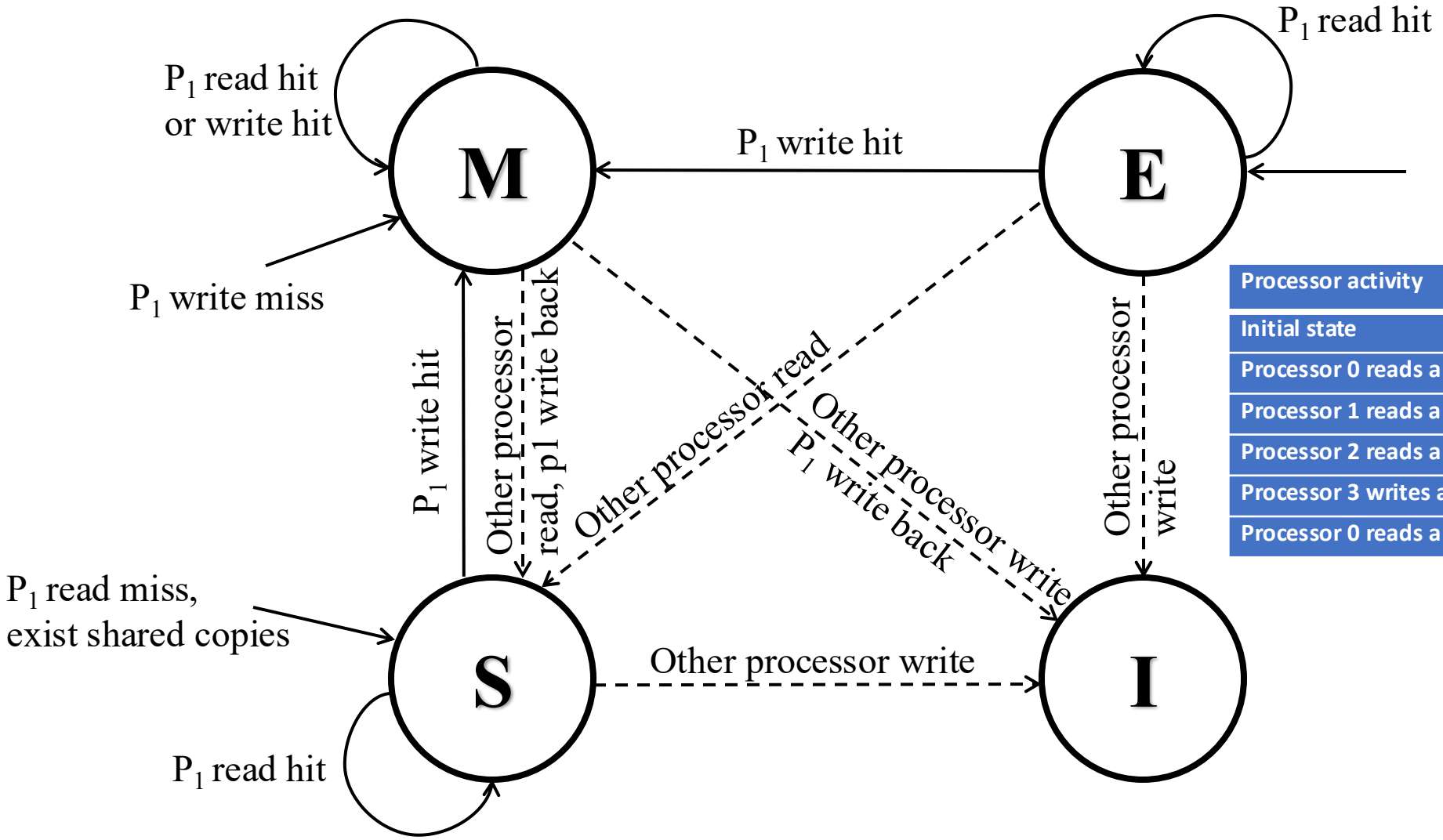


MESI protocol



MESI protocol

Cache state diagram in processor P_1 with the state transitions induced by the local processor shown in solid lines and by the remote processor shown in dashed lines. Local read/write misses are merged to simplify the diagram (For local read miss, if there exist shared copies in other caches, local copy set to S or else set to E).



Processor activity	0	1	2	3
Initial state	I	I	I	I
Processor 0 reads a				
Processor 1 reads a				
Processor 2 reads a				
Processor 3 writes a				
Processor 0 reads a				

MESI protocol

A snapshot of the state associated with 2 caches, on 2 separate cores, in a centralized shared memory system is shown below. In this system, cache coherency is maintained with an MESI snooping protocol. The caches are direct mapped.

P0:

	Tag	Data Word 1	Data Word 2	Data Word 3	Data Word 4	Coherency State
Block 0	1000	10	20	30	40	M
Block 1	4000	500	600	700	800	S
…						
Block N	3000	2	4	6	8	S

P1:

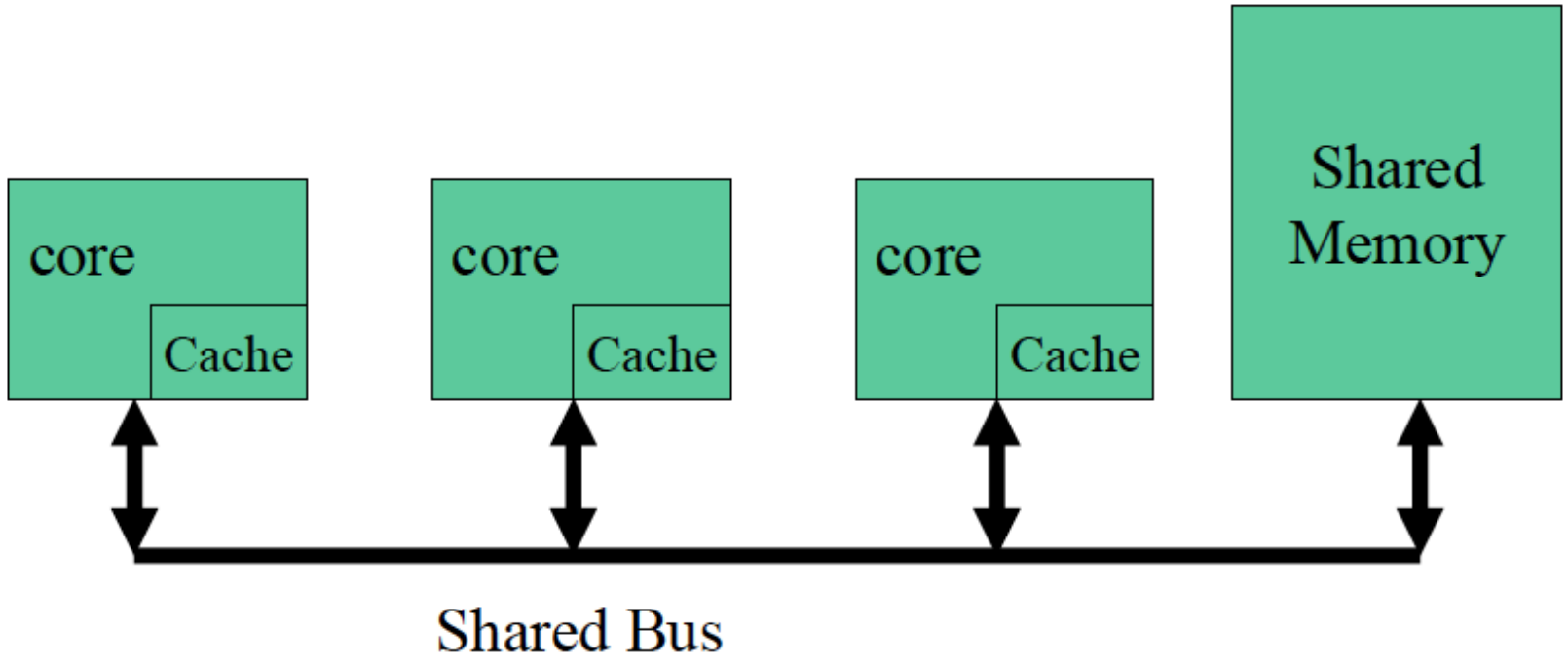
	Tag	Data Word 1	Data Word 2	Data Word 3	Data Word 4	Coherency State
Block 0	1000	10	10	10	10	I
Block 1	8000	500	600	700	800	S
…						
Block N	3000	2	4	6	8	S

If P0 writes to Block 0, what happens to its coherency state?

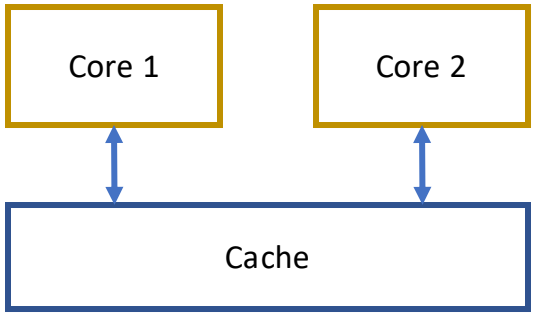
If P1 writes to Block 1, is Block 1 on P0 invalidated?

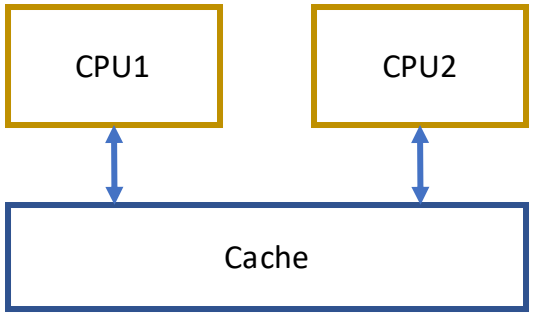
If P1 brings in Block M for reading, and no other cache has a copy, what state is it cached in?

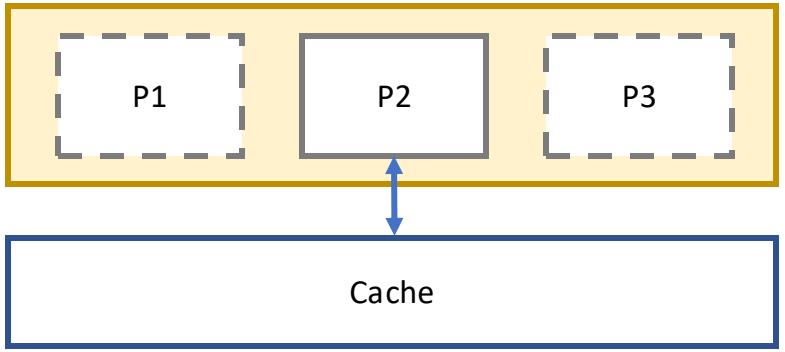
Challenges of Multi-process system based on Shared Memory



Challenges of Multi-process system based on Shared Memory



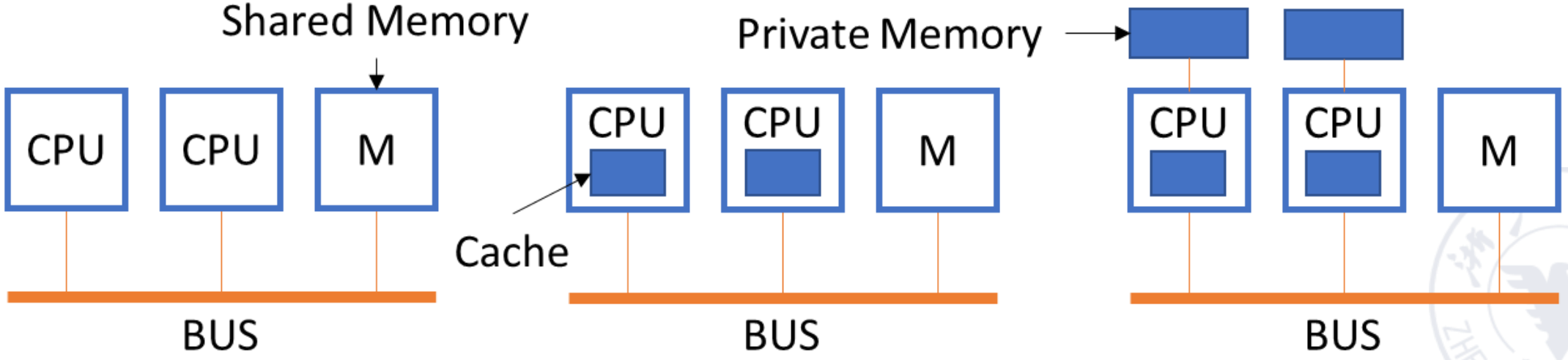




Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

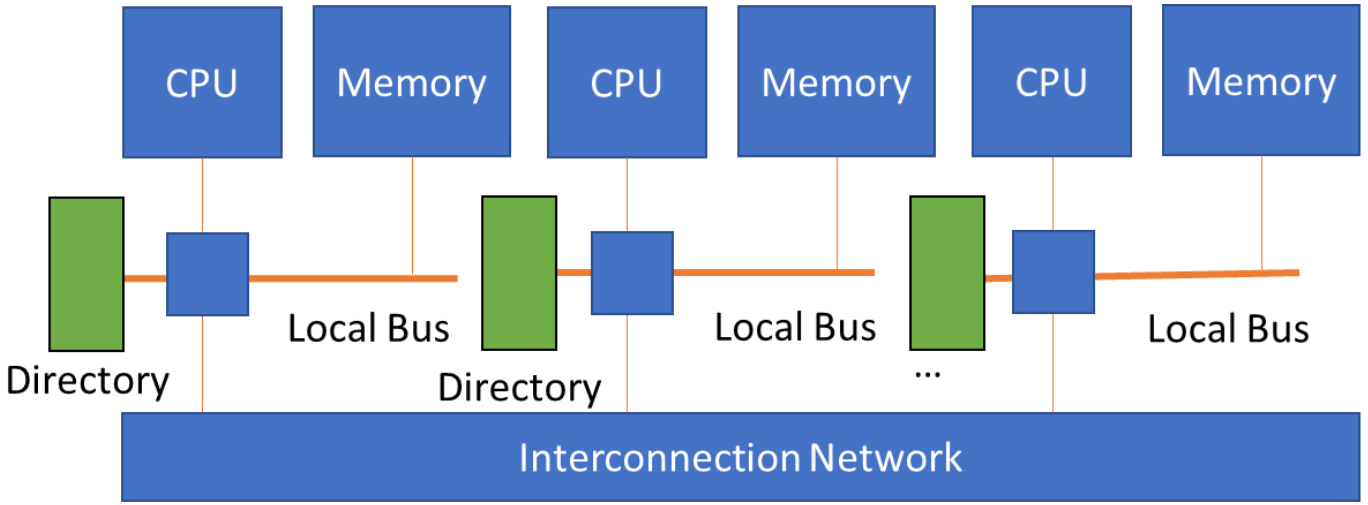
Cache Coherence

- For UMA: *Snoopy coherence protocols*
  - In the snoopy coherence protocols, all processors snoop the bus. When a processor modifies the data in the private cache, it broadcasts invalid information or updated data on the bus to invalidate or update other copies.



Cache Coherence

- For NUMA: *Directory protocol*
  - The directory protocol uses a directory to record which processors in the system have copies of certain storage blocks in the cache. When a processor wants to write a shared block, it sends an invalid signal to those processors that have copies of the block through the directory in a "point-to-point" way, so that all other copies are invalidated.

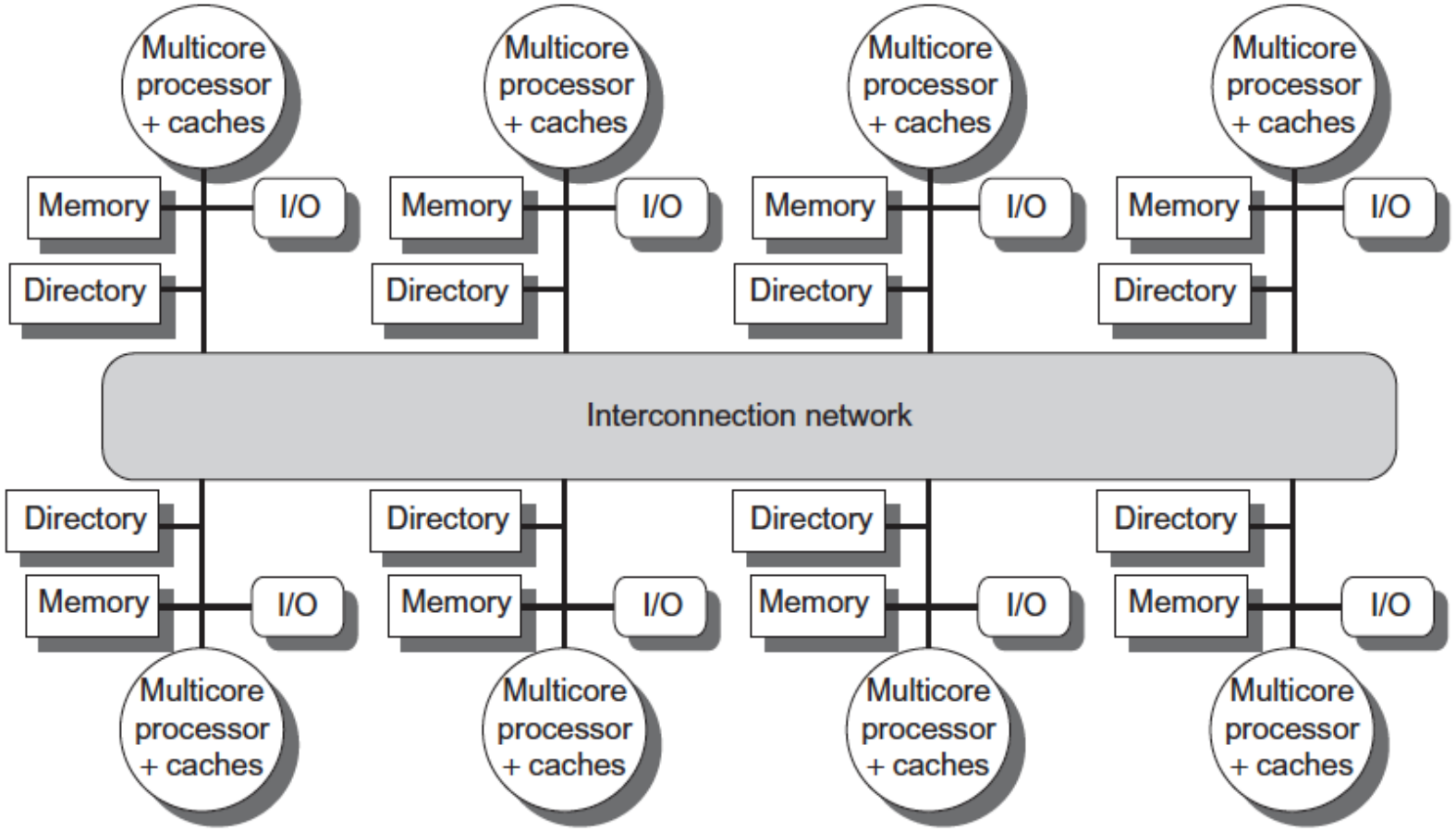


Cache Coherence

- For UMA: *Snoopy coherence protocols*
  - In the snoopy coherence protocols, all processors snoop the bus. When a processor modifies the data in the private cache, it broadcasts invalid information or updated data on the bus to invalidate or update other copies.

- For NUMA: *Directory protocol*
  - The directory protocol uses a directory to record which processors in the system have copies of certain storage blocks in the cache. When a processor wants to write a shared block, it sends an invalid signal to those processors that have copies of the block through the directory in a "point-to-point" way, so that all other copies are invalidated.

Directory protocol



A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.

A directory keeps the state of every block that may be cached. information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

Directory protocol

- For each block, maintain state:
  - *Shared*
    - One or more nodes have the block cached, value in memory is up-to-date (as well as in all caches).
  - *Uncached*
    - No node has a copy of the cache block.
  - *Modified*
    - Exactly one node has a copy of the cache block, and it was written the block, so the memory copy is out-of-date. The processor is called the owner of the block.
- Directory maintains block states and sends invalidation messages.

Directory protocol

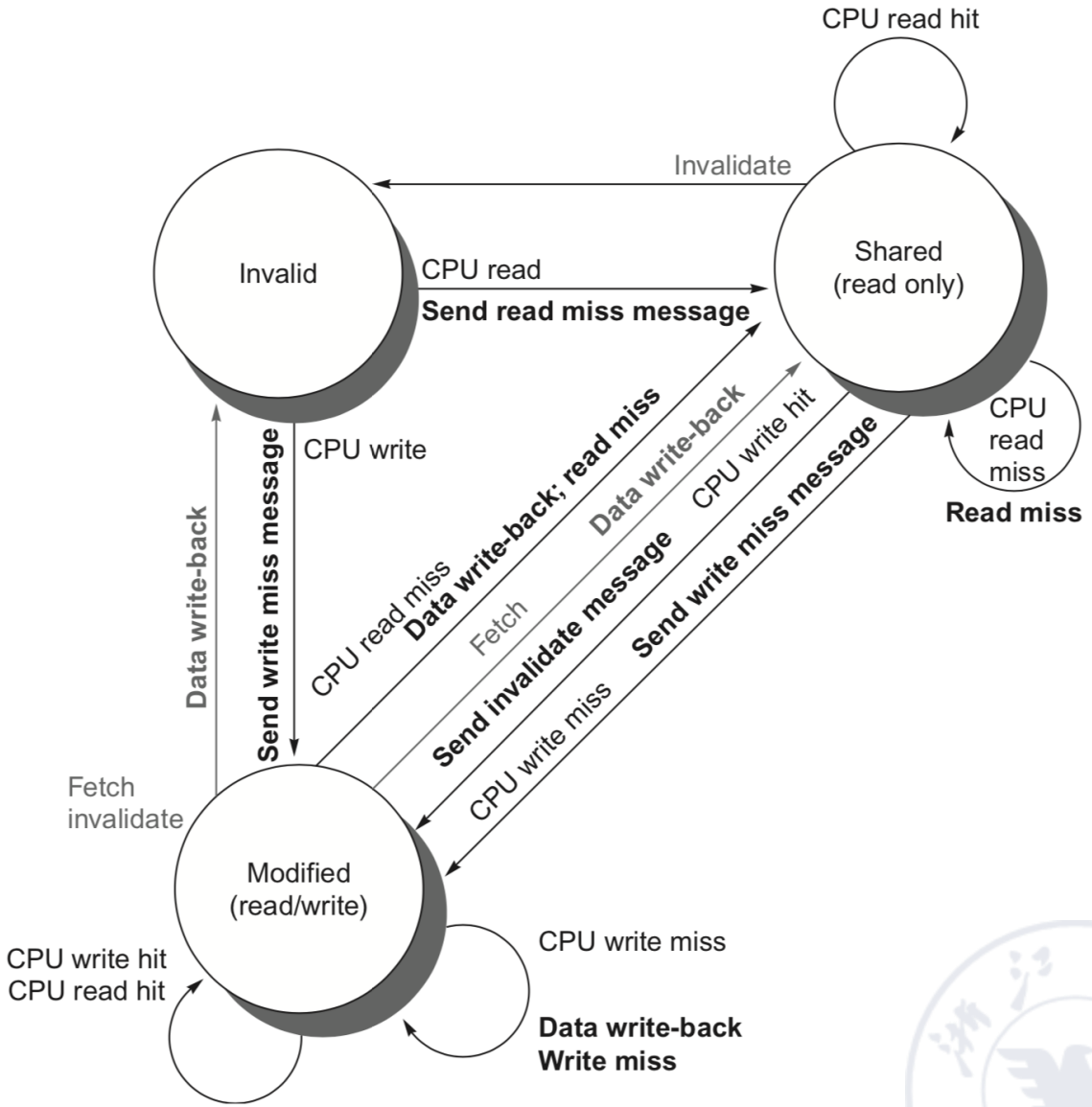
Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/ invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

P=requesting node number
A=requested address
D=data contents

Directory protocol

- State transition diagram for ***an individual cache block*** in a directory-based system.

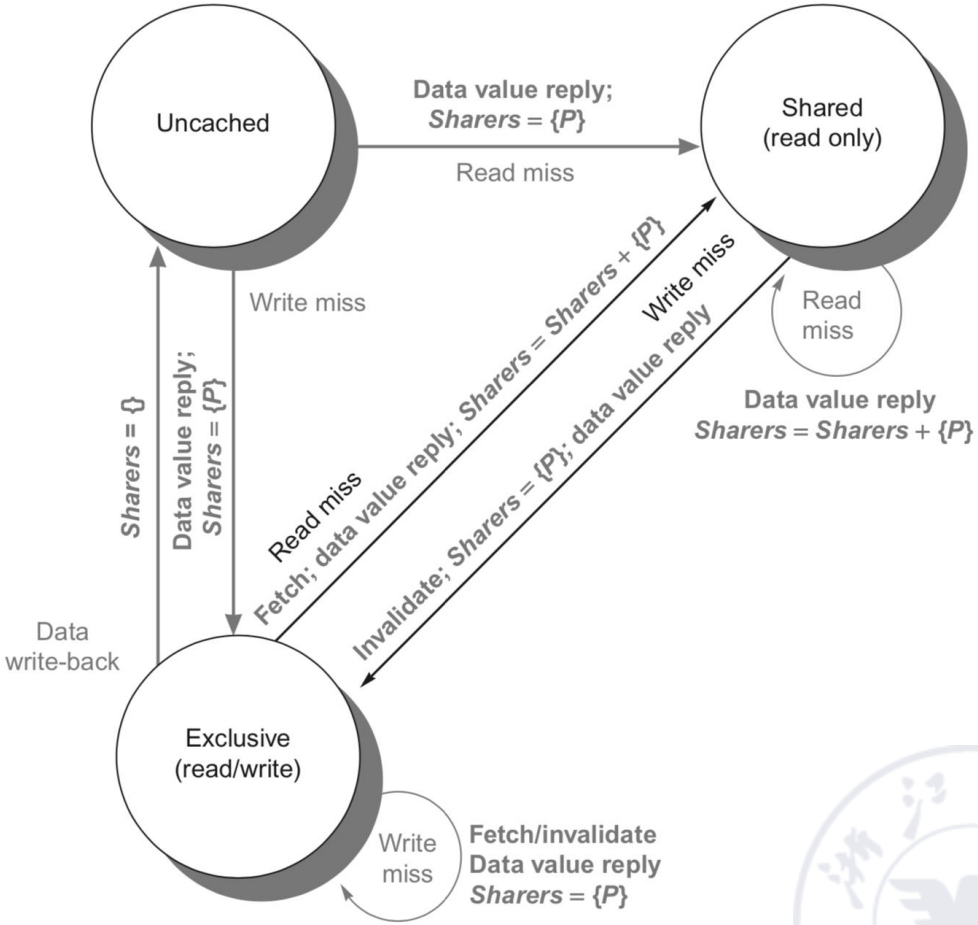
- Requests by the local processor are shown in black, and those from the home directory are shown in gray.



Directory protocol

- The state transition diagram for ***the directory*** has the same states and structure as the transition diagram for an individual cache.

- All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.



Directory protocol

- For uncached block:
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

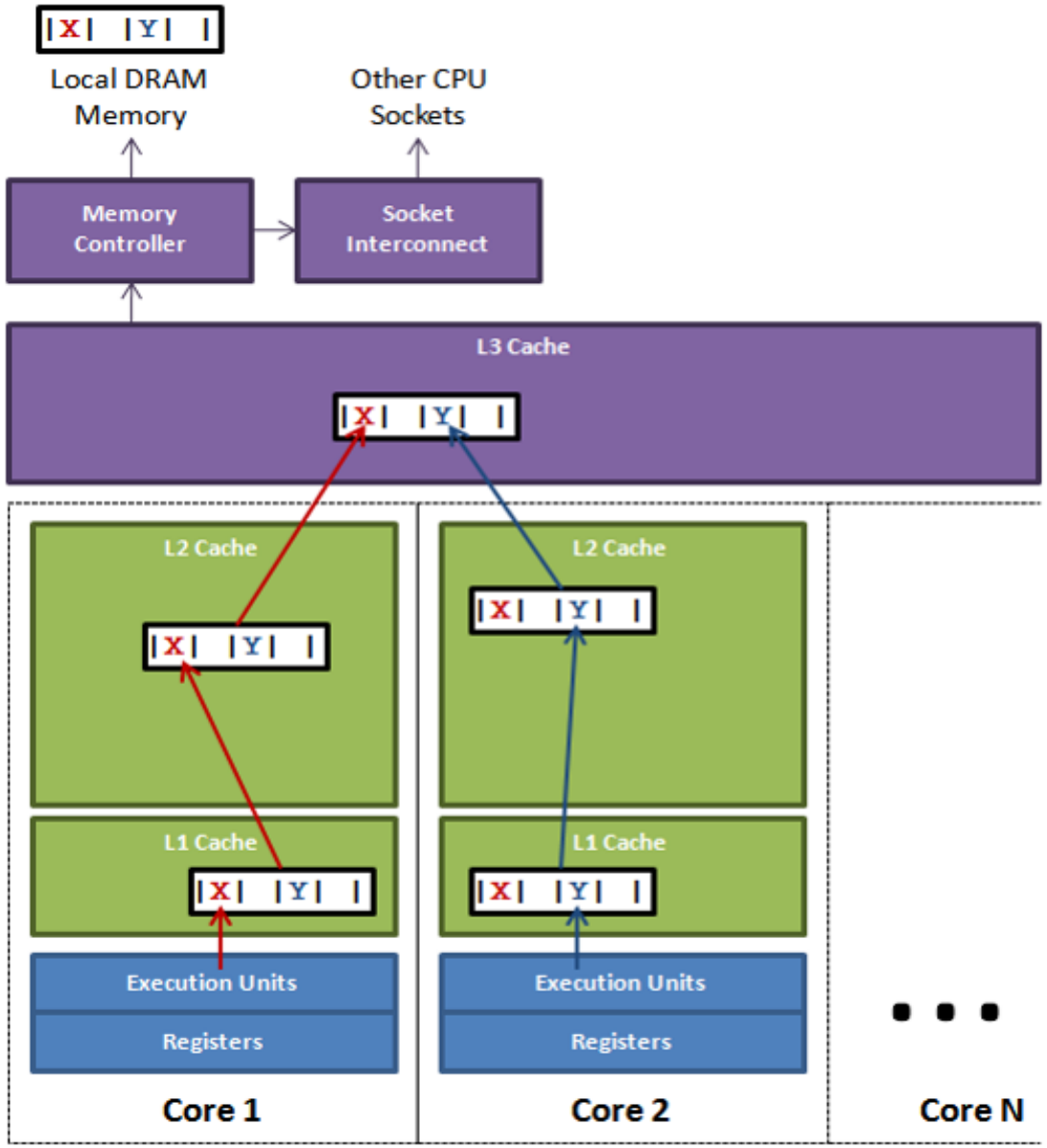
Directory protocol

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

False sharing

```
public class FalseSharingTest {

    public static void main(String[] args) throws InterruptedException
{
        testPointer(new Pointer());
    }

    private static void testPointer(Pointer pointer) throws
InterruptedException {
        long start = System.currentTimeMillis();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.x++;
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.y++;
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(System.currentTimeMillis() - start);
        System.out.println(pointer);
    }
}

class Pointer {
    volatile long x;
    volatile long y;
}
```

False sharing



Avoid False sharing

I.

```
class Pointer {
    volatile long x;
    long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
}
```

II.

```
class Pointer {
    MyLong x = new MyLong();
    MyLong y = new MyLong();
}

class MyLong {
    volatile long value;
    long p1, p2, p3, p4, p5, p6, p7;
}
```

III.

```
@sun.misc.Contended
class MyLong {
    volatile long value;
}
```

Memory Consistency and Cache Coherence

- Memory Consistency Need Memory Consistency Model
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- Cache Coherence Need Cache Coherence Protocol
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors
  - Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. This is because correct coherence ensures that the caches never enable new or different *functional* behavior.

Memory Consistency

What properties must be enforced among reads and writes to different locations by different processors?

- Example:

Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

- Sequential consistency (reduces potential performance):
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Relaxed Consistency Models

- Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering

Relaxed Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - “Total store ordering” or processor consistency
  - Relax W → W and W → R
    - “Partial store order”
  - Relax R → W, R → R, W → W and W → R
    - “Weak ordering” and “release consistency”

MIMD Architecture

- Different memory access models of MIMD multiprocessor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

MPP

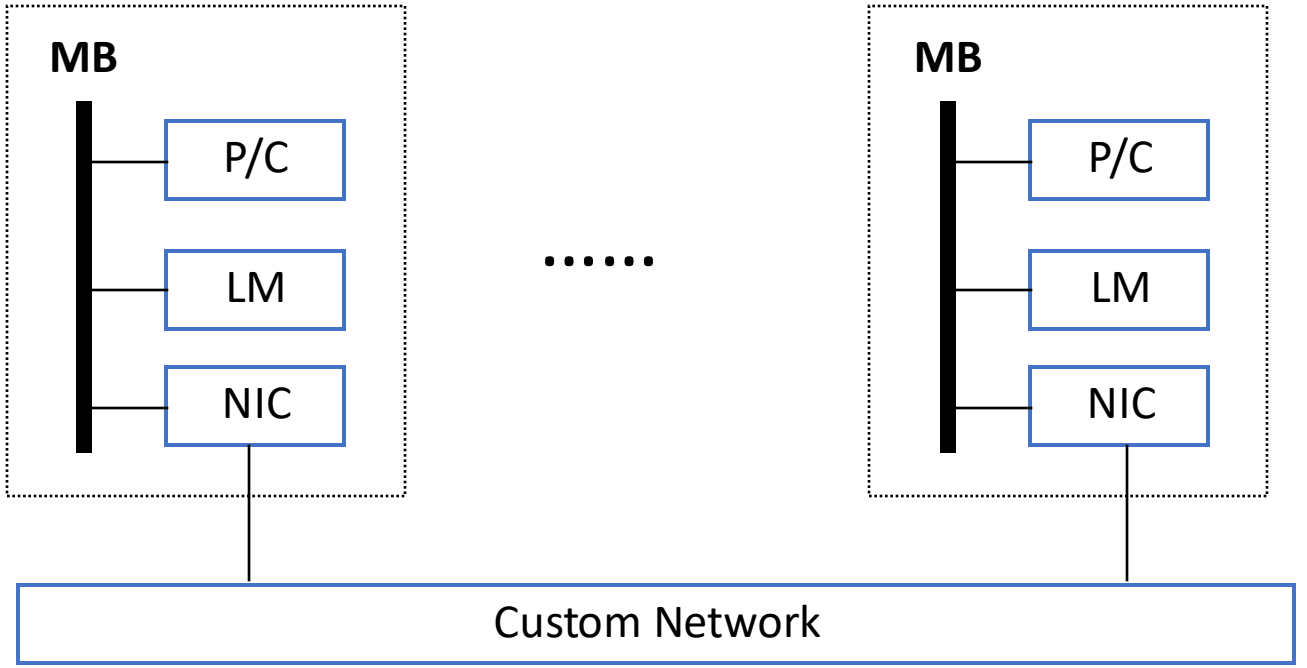
- The MPP system is a massively parallel computer system composed of hundreds of processors.
- In the past, it was mainly used for calculation-oriented occasions such as scientific calculation and engineering simulation, but it is also widely used in commercial and network applications.
- Development is difficult, the price is high, and the market is limited. It is a symbol of the country's comprehensive strength.

Characteristics of MPP

- MPP systems generally use standard commercial CPUs as their processors.
- The MPP system uses a high-performance private interconnection network, which can deliver messages with low latency and high bandwidth.
- The MPP system has powerful input/output capabilities.
- The MPP system is capable of special fault-tolerant processing.

MPP

LM： Local Memory
NIC： Network interface circuit
MB： Memory Bus



MIMD Architecture

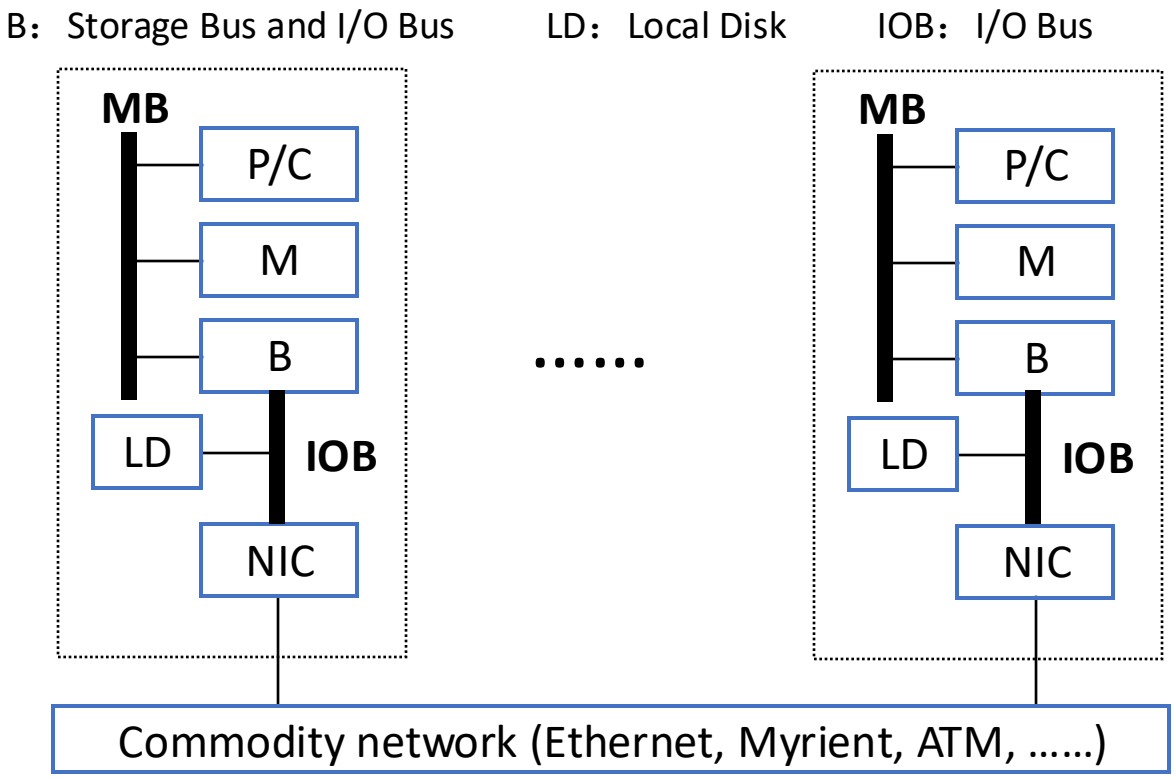
- Different memory access models of MIMD multiprocessor system
  - Uniform Memory Access (UMA)
  - Non Uniform Memory Access (NUMA)
  - Cache Only Memory Access (COMA)

- Further division of MIMD multi-computer system
  - Massively Parallel Processors (MPP)
  - Cluster of Workstations(COW)

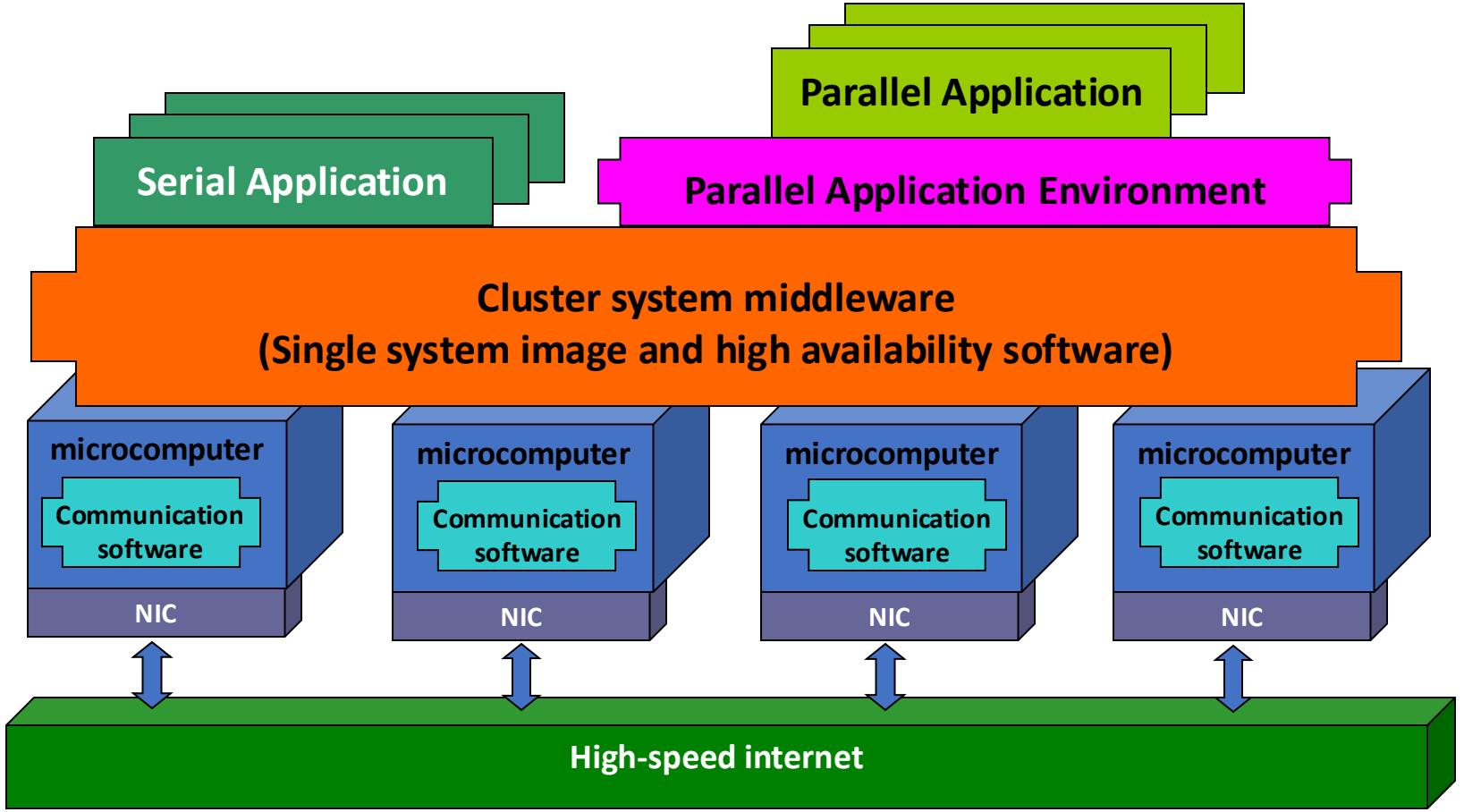
COW

- The COW system is composed of a large number of PCs or workstations connected together through a commercial network.
- COW can be assembled completely using commercially available components. These commercial components are mass-produced products, so they can achieve higher cost performance.
- There are two main types of COW that dominate: centralized and decentralized.

COW


B：Storage Bus and I/O Bus
LD：Local Disk
IOB：I/O Bus
MB
P/C
M
B
LD
IOB
NIC
.....
MB
P/C
M
B
LD
IOB
NIC
Commodity network (Ethernet, Myrient, ATM, ……)

COW Architecture

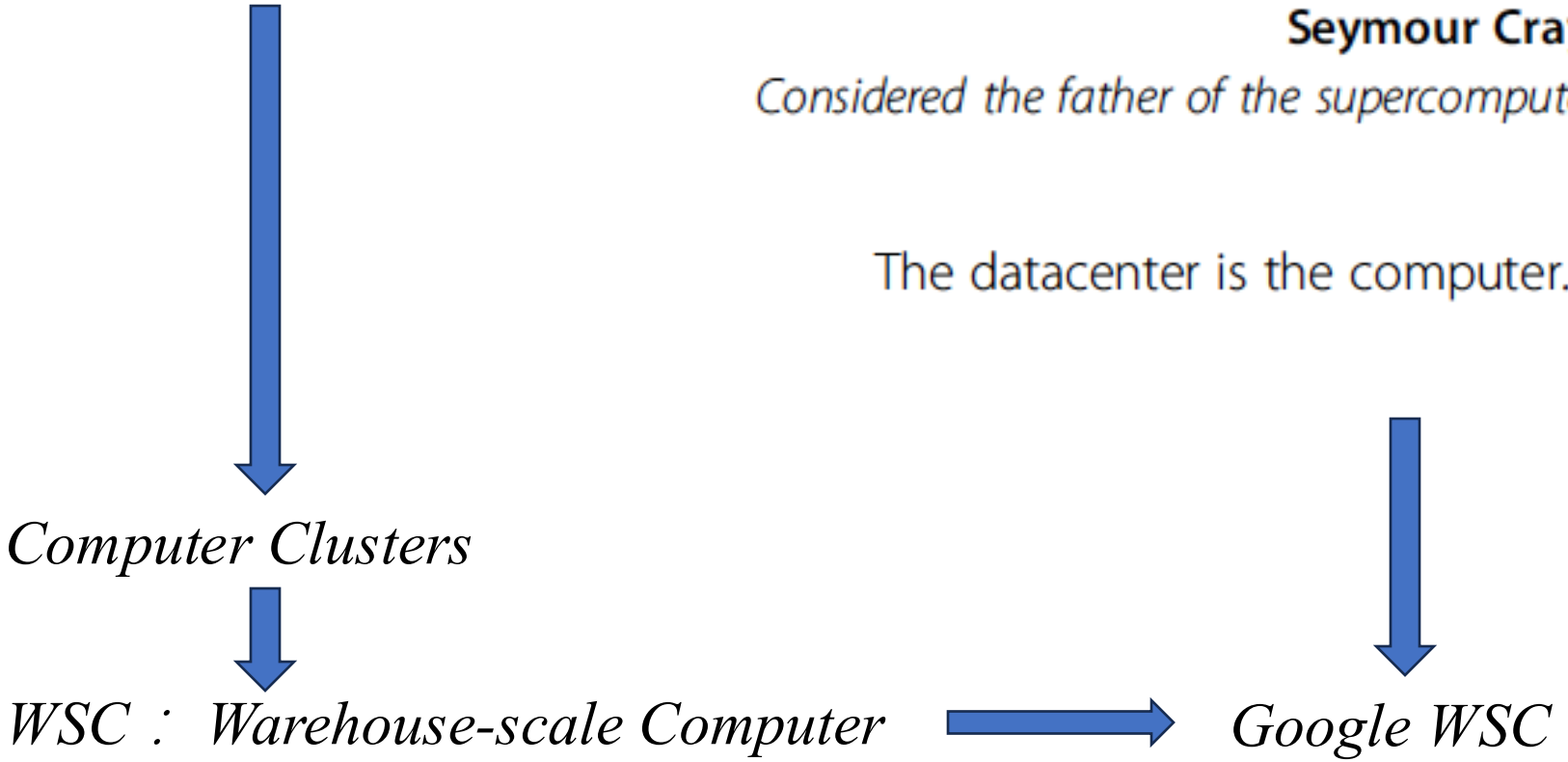


Anyone can build a fast CPU. The trick is to build a fast system.

Seymour Cray,
Considered the father of the supercomputer

The datacenter is the computer.

Luiz André Barroso,
Google (2007)



Domain-Specific Architectures

- Moore's Law enabled:
  - Deep memory hierarchy
  - Wide SIMD units
  - Deep pipelines
  - Branch prediction
  - Out-of-order execution
  - Speculative prefetching
  - Multithreading
  - Multiprocessing

- Objective:
  - Extract performance from software that is oblivious to architecture

Guidelines for DSAs

- Use dedicated memories to minimize data movement
- Invest resources into more arithmetic units or bigger memories
- Use the easiest form of parallelism that matches the domain
- Reduce data size and type to the simplest needed for the domain
- Use a domain-specific programming language

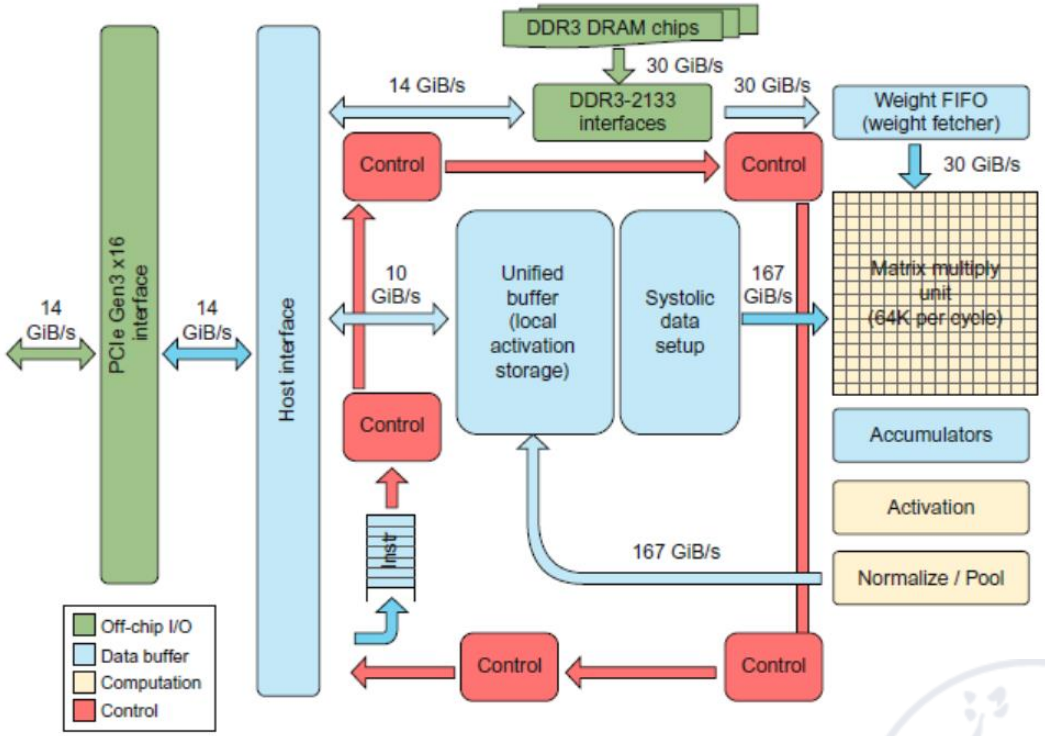
Example: Convolutional Neural Network

Abstraction in Computer Architecture View

- Batches:
  - Reuse weights once fetched from memory across multiple inputs
  - Increases operational intensity
- Quantization
  - Use 8- or 16-bit fixed point
- Summary:
  - Need the following kernels:
    - *Matrix-vector multiply*
    - *Matrix-matrix multiply*
    - *ReLU*
    - *Sigmoid*
    - *…*

Tensor Processing Unit

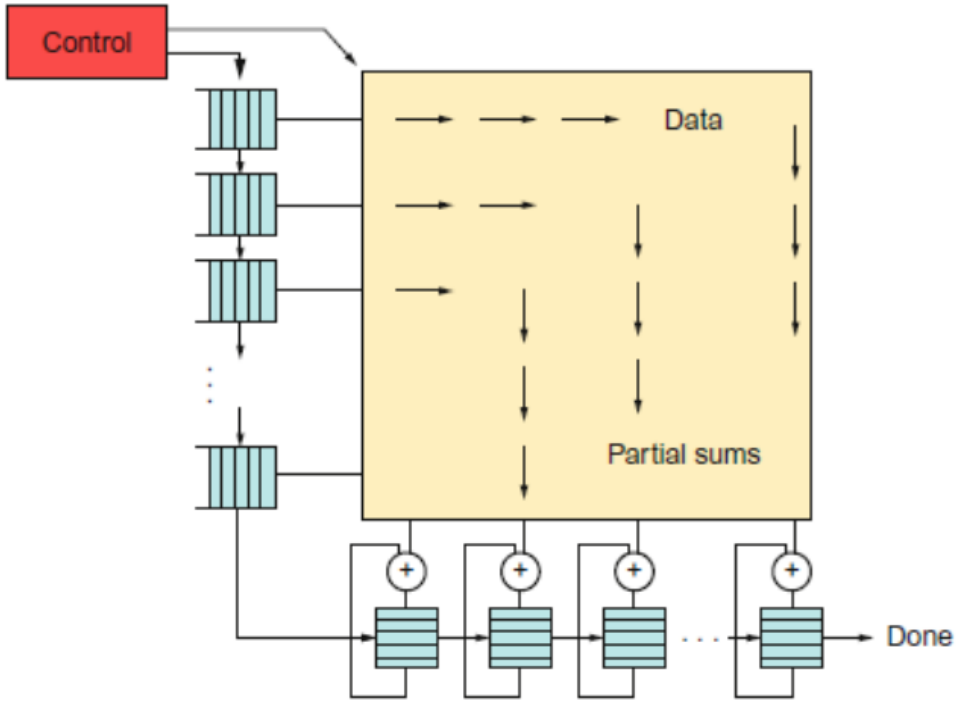
- Google's DNN ASIC
- 256 x 256 8-bit matrix multiply unit
- Large software-managed scratchpad
- Coprocessor on the PCIe bus

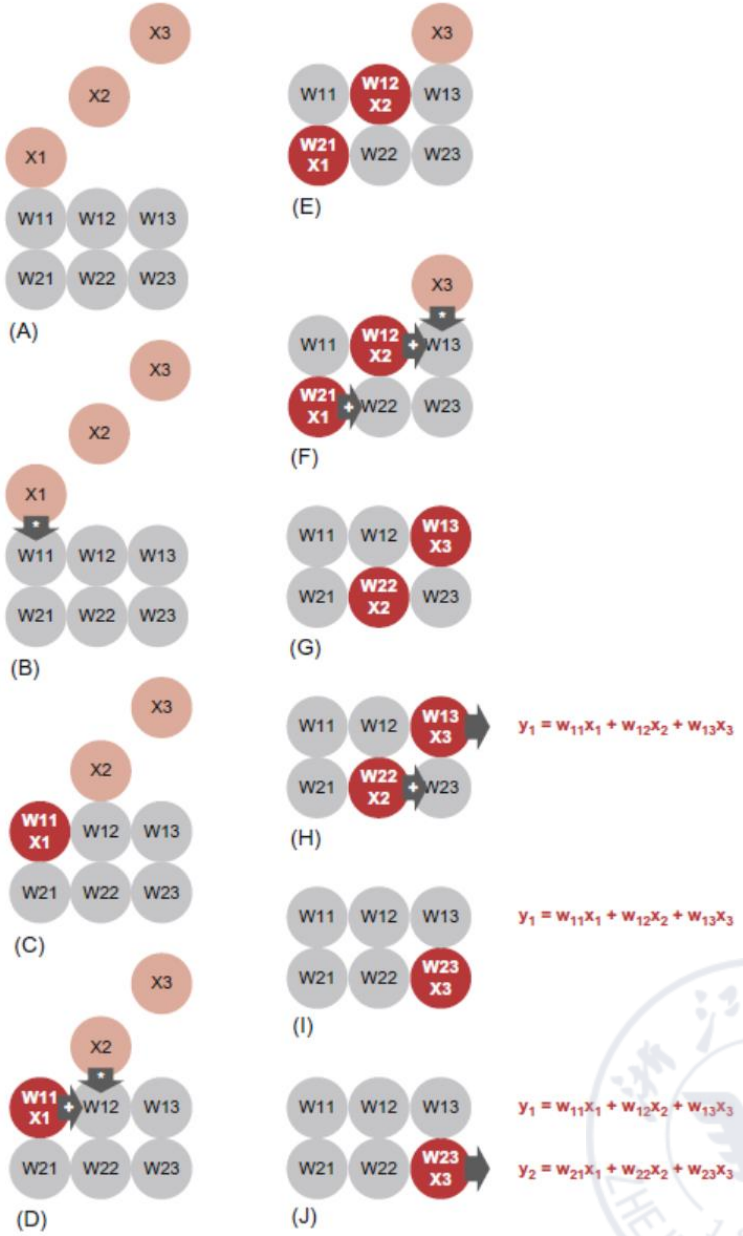


TPU ISA

- **Read_Host_Memory**
  - Reads memory from the CPU memory into the unified buffer
- **Read_Weights**
  - Reads weights from the Weight Memory into the Weight FIFO as input to the Matrix Unit
- **MatrixMatrixMultiply/Convolve**
  - Perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the accumulators
  - takes a variable-sized B*256 input, multiplies it by a 256x256 constant input, and produces a B*256 output, taking B pipelined cycles to complete
- **Activate**
  - Computes activation function
- **Write_Host_Memory**
  - Writes data from unified buffer into host memory

Matrix Calculations in TPU

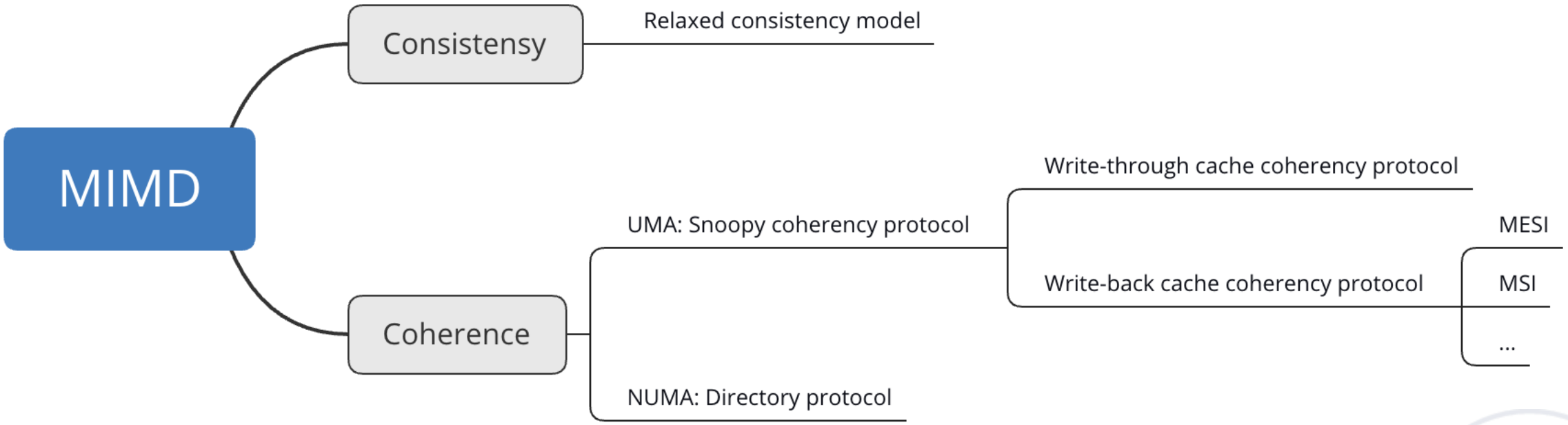




The TPU and the Guidelines

- Use dedicated memories
  - 24 MiB dedicated buffer, 4 MiB accumulator buffers
- Invest resources in arithmetic units and dedicated memories
  - 60% of the memory and 250X the arithmetic units of a server-class CPU
- Use the easiest form of parallelism that matches the domain
  - Exploits 2D SIMD parallelism
- Reduce the data size and type needed for the domain
  - Primarily uses 8-bit integers
- Use a domain-specific programming language
  - Uses TensorFlow

Summary


MIMD
Consistensy
Relaxed consistency model
Coherence
UMA: Snoopy coherency protocol
Write-through cache coherency protocol
Write-back cache coherency protocol
MESI
MSI
...
NUMA: Directory protocol