9 Multiprocessor / Multicore / Multicomputer Systems

To enhance system performance and (in some cases) to increase availability, multiple processing units work in parallel.

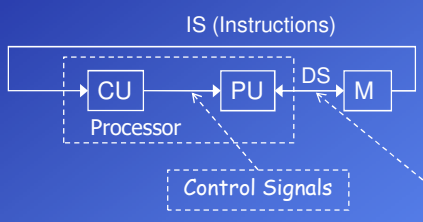
Levels of parallelism in software:

- Instruction-level parallelism: Pipelining. Portions of different instructions run in parallel.
- Parallel programming: Portions of a single program (task) run on multiple processors simultaneously (a type of thread-level parallelism).
- Job-level or process-level parallelism : Independent applications run on different processors (also a type of thread-level parallelism).
- Data-level parallelism: Data pipelining or multiple functional units, e.g arithmetic logic units (ALU). For example different elements of an array can be processed simultaneously.

9.1 Flynn's Taxonomy by Michael J. Flynn (1934-)

There are different types of parallel computers.
A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability.

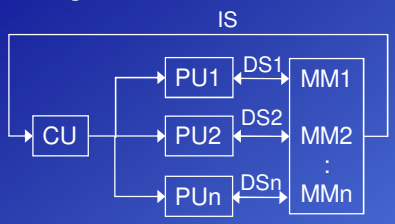
a) SISD: Single Instruction stream, Single Data stream



CU: Control Unit
PU: Processing Unit
M: Memory
IS: Instruction Stream
DS: Data Stream

A single processor executes a single instruction on a single piece of data at a time (uniprocessor).

b) SIMD: Single Instruction stream, Multiple Data stream



CU: Control Unit
PU: Processing Unit
MM: Memory Module

Each PU can have its own memory module.

Several PUs are under the control of the same CU. All PUs receive the same instruction from the CU.

Each PU has its own data memory (hence multiple data), so that the same instruction is executed by multiple PUs using different data streams.

Example: Vector and array processors.

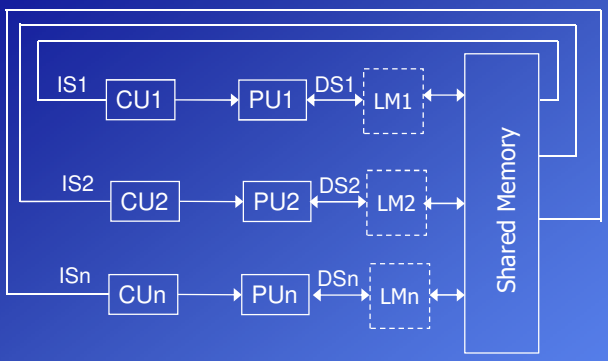
c) MISD: Multiple Instruction stream, Single Data stream (No commercial multiprocessor of this type has been built to date.)

Different instructions are executed on the same data at the same time.

It can be used to provide fault tolerance with different backup systems operating on the same data to provide independent results that are compared to each other.

d) MIMD: Multiple Instruction stream, Multiple Data stream

i. with shared memory



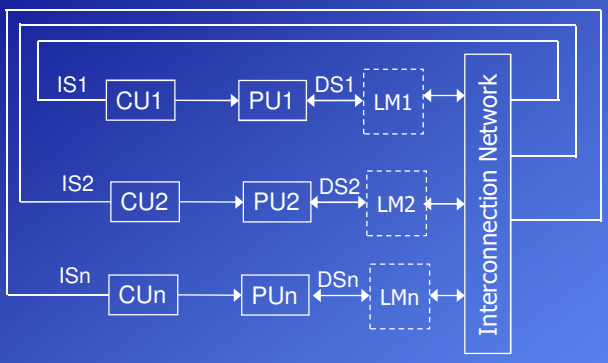
CU: Control Unit
PU: Processing Unit
LM: Local Memory
IS: Instruction Stream
DS: Data Stream

A set of processors simultaneously execute different instruction sequences on different data sets.

- **Shared memory** (*tightly coupled*) systems: The processors share a common memory, and they communicate with each other (share data) through that memory.

The processors may also have their local memories (cache).

d) MIMD: Multiple Instruction stream, Multiple Data stream (cont'd)
ii. with distributed memory

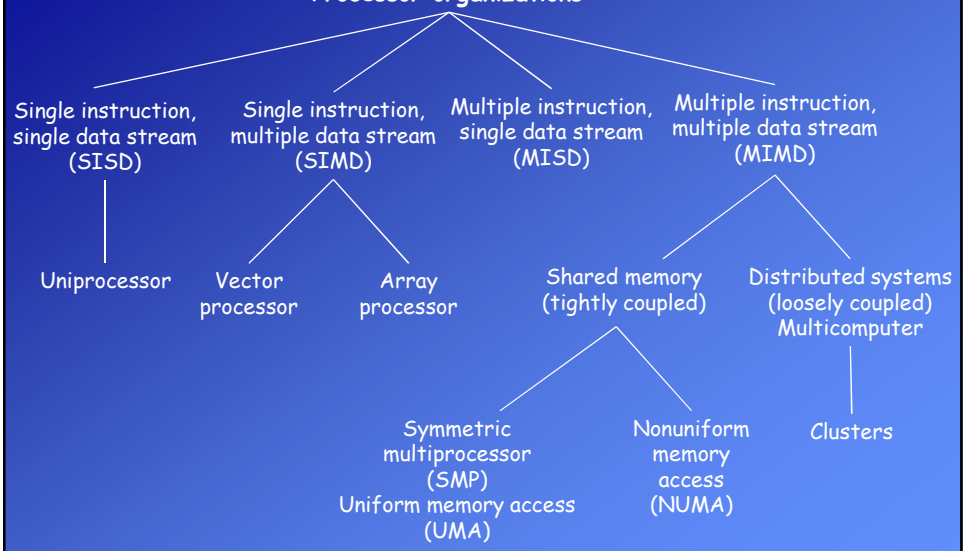


CU: Control Unit
PU: Processing Unit
LM: Local Memory
IS: Instruction Stream
DS: Data Stream

- **Distributed memory** (loosely coupled) systems: A collection of independent uniprocessors are interconnected to form a cluster.

Communication among the computers is either via fixed paths or via some network facility.

Processor organizations



9.2 Shared Memory (Tightly Coupled) Systems

- A shared memory multiprocessor offers the programmer a single physical address space (shared memory).
- Processors communicate through shared variables in memory.
- All processors are capable of accessing any memory location via load and store instructions.
- The system is controlled by an integrated common operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.
- Because of shared variables, the operating system must support synchronization among processors (processes, threads).
- There are two different types of shared memory systems:
  a) Symmetric multiprocessor (SMP) or Uniform memory access (**UMA**) systems:
     It takes about the same time to access main memory (symmetric) no matter which processor requests it and no matter which word is requested.
  b) Nonuniform memory access (**NUMA**) multiprocessors:
     The processors still share the same single address space, but memory modules are physically distributed in the system.
     A processor can access nearby memory faster.

9.2.1 Symmetric Multiprocessors (SMP) / Uniform memory access (UMA) systems

Characteristics:

- Processors have access to a single, common address space (shared memory) and are controlled by a single operating system.
- There are two or more processors with identical capabilities.
- All processors can perform the same functions (symmetric).
- Processors share the same main memory and I/O facilities.
- System components are interconnected by a bus or other internal connection scheme such as a crossbar switch.
- The memory access time is approximately the same for each processor (symmetric) (UMA).

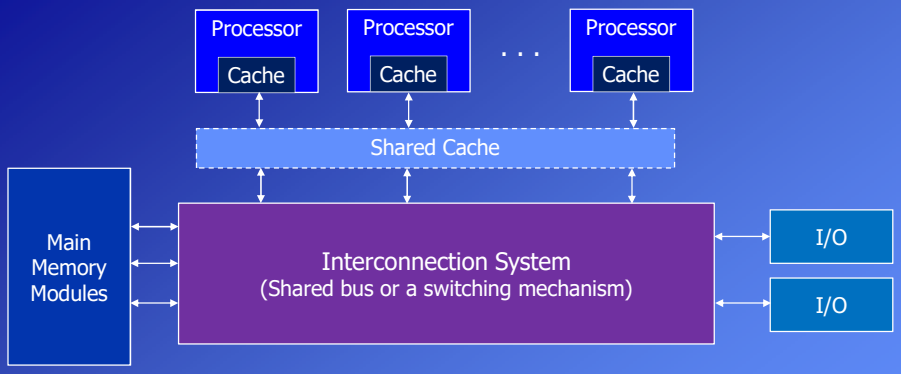
9.2.1 Symmetric Multiprocessors (SMP) / Uniform memory access (UMA) systems (cont'd)

Potential benefits:

- Performance: In situations where more than one program executes at the same time, an SMP system will have considerably better performance than a uniprocessor because different programs can run on different processors simultaneously.
- Availability: Since all processors can perform the same functions, the failure of a single processor does not halt the machine.
- Incremental growth (scaling): A user can increase the performance of a system by adding another processor.

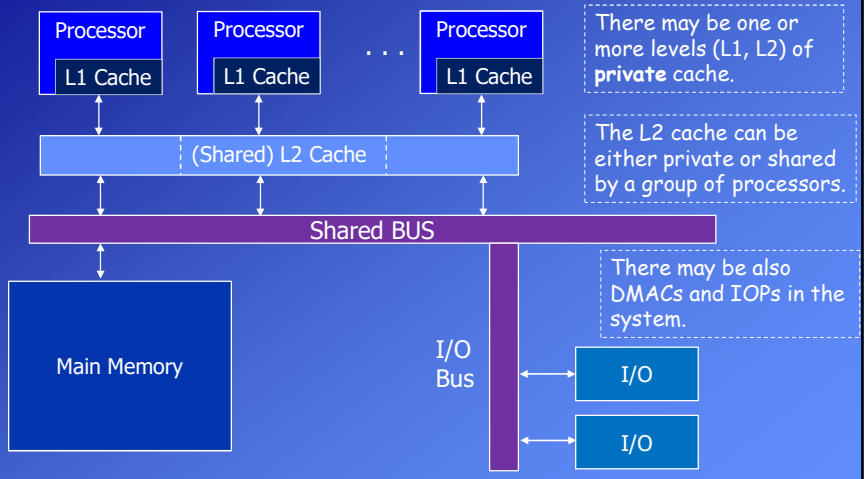
  However! As more processors are added, competition for access to the bus leads to a decline in performance (64 processors max.).

Organization (SMP, UMA):



- Each processor consists of a control unit, ALU, registers, and, typically, one or more levels of cache.
- The memory can be interleaved or a multiported, allowing simultaneous accesses to separate blocks of memory.
- The interconnection system can be designed in different ways (e.g., a shared bus or a crossbar switch).

Symmetric Multiprocessor Organization Using a Shared Bus:



Time-sharing: When one module is controlling the bus, other modules are locked out and must, if necessary, suspend operation until bus access is achieved.
Bus arbitration is necessary.

Advantages:

- **Simplicity:** The physical interface and the addressing, arbitration, and time-sharing logic of each processor remain the same as in a single-processor system.
- **Flexibility:** It is generally easy to expand the system by attaching more processors to the bus (but, there is a limit).
- **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

Drawback:

- **Performance:**
  - All memory references pass through the common bus.
  - The bus cycle time limits the speed of the system.
  - The common bus is used on a time-sharing basis. When a processor or DMAC is accessing the bus, other processors cannot access main memory.
  - The shared bus limits the number of processors in the system to 16-64.

Solution:

- Equip each processor with a local cache memory: Most frequently used data are kept in cache memories. Hence, the need to access the main memory is reduced.
  - **Cache coherence** problem: If a word is modified in one cache, the copies of the same word in other caches will be invalid. Other processors must be alerted that an update has taken place (explained in chapter 9.4 Cache Coherence).

9.2.2 Nonuniform memory access (NUMA) multiprocessors

In SMP systems, the common bus is a performance bottleneck.

The number of processors is limited.

Loosely coupled systems (clusters) can be a solution, but in these systems, applications cannot see a global memory.

NUMA systems are designed to achieve large-scale multiprocessing while retaining the advantages of shared memory.

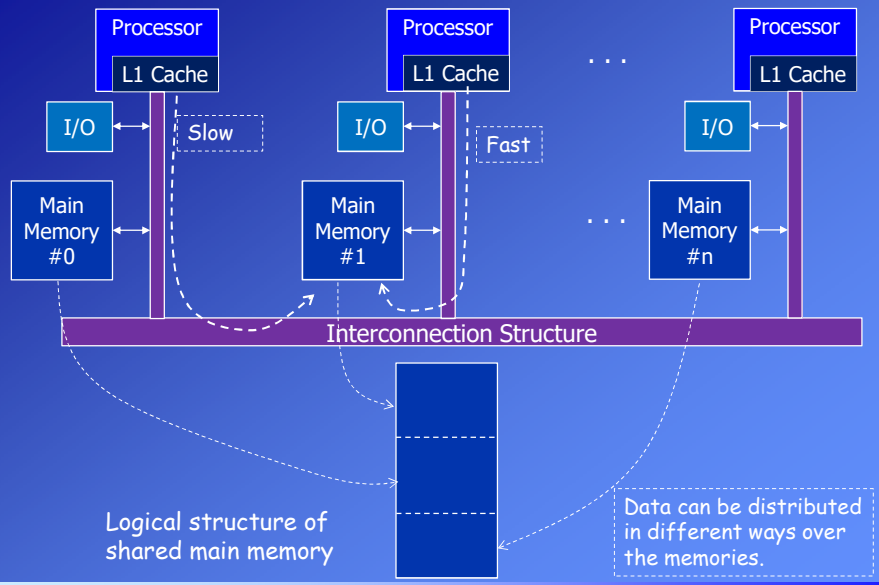
Characteristics:

- Processors have access to a single address space (shared memory) and are controlled by a single operating system.
- The shared memory is physically distributed to all CPUs. These systems are also called **distributed shared memory** systems.
- A CPU can access its own memory module faster than other modules.

Performance:

- If processes and data can be distributed in the system so that CPUs are mostly accessing their own main memory modules (or local cache memories) and rarely remote memory modules, then the performance of the system increases.
- Spatial and temporal locality of programs and data play an important role again.

A NUMA Multiprocessor Organization:



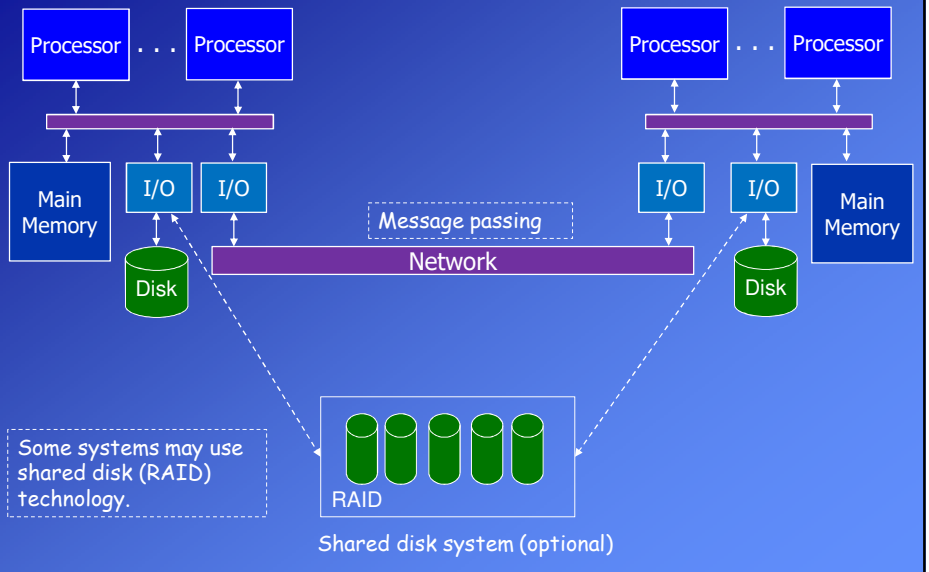
9.3 Distributed (loosely coupled) systems, Multicomputers

- Each processor has its own physical address space.
- These processors communicate via message passing.
- The most widespread example of the message passing system are **clusters**.
- Clusters are collections of computers that are connected to each other over standard network equipment.
- When these clusters grow to tens of thousands of servers and beyond, they are called warehouse-scale computers (cloud computing).

Benefits:

- **Scalability:**
  - A cluster can have tens, hundreds, or even thousands of machines, each of which is a multiprocessor.
  - It is possible to add new systems to the cluster in small increments.
- **High availability:**
  - Each node in a cluster is a standalone computer; therefore, the failure of one node does not mean loss of service.
- **Superior price/performance:**
  - Using cheap commodity building blocks, it is possible to build a cluster with great computing power.

A Loosely Coupled (distributed) System Organization:

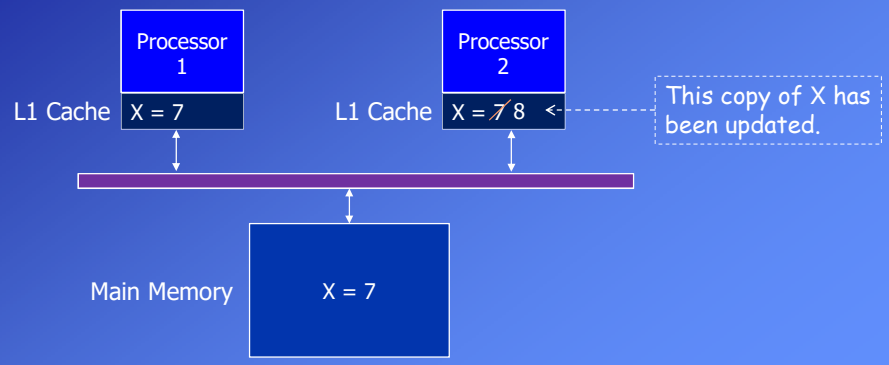


9.4 Cache Coherence

To reduce the average access time and the required memory bandwidth, cache memories are used.

Caching of shared data introduces the **cache coherence problem**.

Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.



9.4.1 Software solutions:

- Software cache coherence schemes attempt to avoid the need for additional hardware circuitry.
- The compiler and operating system deal with the problem at compile time.
- However, they make conservative decisions, leading to inefficient cache utilization.
- Compiler-based mechanisms perform an analysis on the code to determine which data items may become (when) unsafe for caching, and they mark those items.

  The operating system or hardware then prevents these items from being cached.
- The simplest approach is to prevent any shared data variables from being cached (too conservative and inefficient).
- The more efficient approach is to analyze the code to determine safe and critical periods for shared variables and to insert instructions into the code to enforce cache coherence.

9.4.2 Hardware solutions:

a) Directory protocols:

There is a centralized controller that maintains a directory that is stored in main memory.

The directory contains information about which processors have a copy of which lines (frames) in their private caches.

- Writing to (updating) cache:

  When a processor wants to write to a local copy of a line, it must request exclusive access to the line from the controller.

  The controller sends a message to all processors, forcing each of them to invalidate its copy.

  After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor.

a) Directory protocols (cont'd):

- Reading:

  When a processor tries to read a line that is exclusively granted to another processor, a miss occurs (data is invalid).

  If the write-through mechanism is used, the data in main memory is valid.

  If the write-back mechanism is used, the controller issues a command to the processor holding that line that requires the processor to do a write back to main memory.

  The line may now be shared for reading by the original processor and the requesting processor.

Drawbacks:

- The centralized controller is a bottleneck. All requests are sent to the same controller.
- Overhead of communication between local cache controllers and the central controller.

Advantage:

- Effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

b) Snoopy protocols:

- The responsibility for maintaining cache coherence is **distributed** among all of the cache controllers in the multiprocessor system.
- When a shared cache frame (line) is updated, the local controller announces this operation to all other caches by a broadcast mechanism.
- Each cache controller is able to "snoop" on the network to observe these broadcasted notifications, and react accordingly (for example, invalidate the copy).
- Snoopy protocols are suitable for a bus-based multiprocessor because the shared bus provides a simple mechanism for broadcasting and snooping.
- Remember: Local caches are used to decrease the traffic on the shared bus. Therefore, care must be taken not to increase the traffic on the shared bus by broadcasting and snooping.

b) Snoopy protocols (cont'd):

There are two types of snoopy protocols: write-invalidate and write-update

Write-invalidate protocol:

- When one of the processors wants to perform a write to the line in the private cache, it sends an "invalidate" message.
- All snooping cache controllers invalidate their copies of the appropriate cache line.
- Once the line is exclusive (not shared), the owning processor can write to its copy.
- If the write-through method is used, the data is also written to main memory.
- If another CPU attempts to read this data a miss occurs and data is fetched from main memory.

Write-update protocol:

- When one of the processors wants to update a shared line, it broadcasts the new data to all other processors so that they can also update their private caches.
- At the same time, the CPU updates its own copy in the cache.

Experience has shown that **invalidate protocols use significantly less bandwidth**.

The MESI (Modified Exclusive Shared Invalid) Protocol

- A snoopy, write-invalidate cache coherence protocol
- It allows the use of the write-back method. Main memory is not updated until it is necessary to replace the frame.
- Each cache frame (line) can be in one of four states (2 status bits):

M (Modified): The frame in this cache is modified. It is different from the main memory.
This frame is valid only in this cache.

E (Exclusive): The frame in the cache is the same as that in main memory and is not present in any other cache.

S (Shared): The frame in the cache is the same as that in main memory and may be present in another cache.

I (Invalid): The line in the cache does not contain valid data.

	Modified	Exclusive	Shared	Invalid
Is the cache frame valid?	Yes	Yes	Yes	No
Is the data in the main memory valid?	No	Yes	Yes	-
Do copies exist in other caches?	No	No	Maybe	Maybe

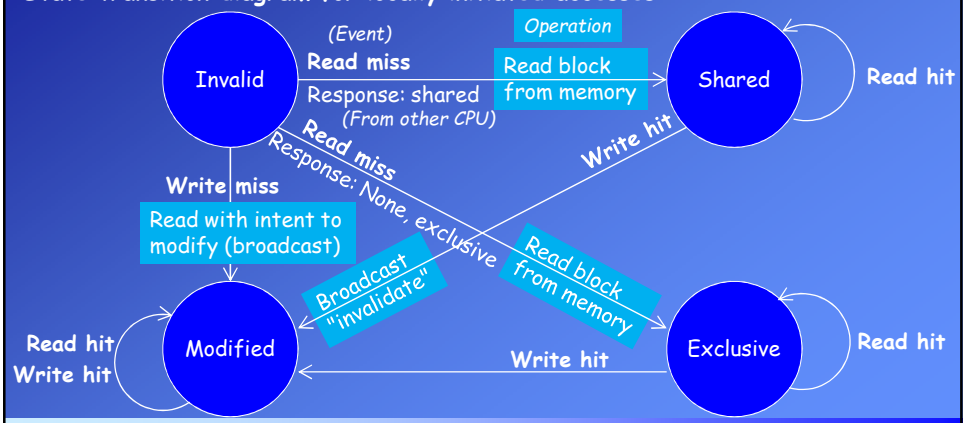
State transitions of frames:

Two bits are assigned to each cache frame in tag memory to keep the state value.

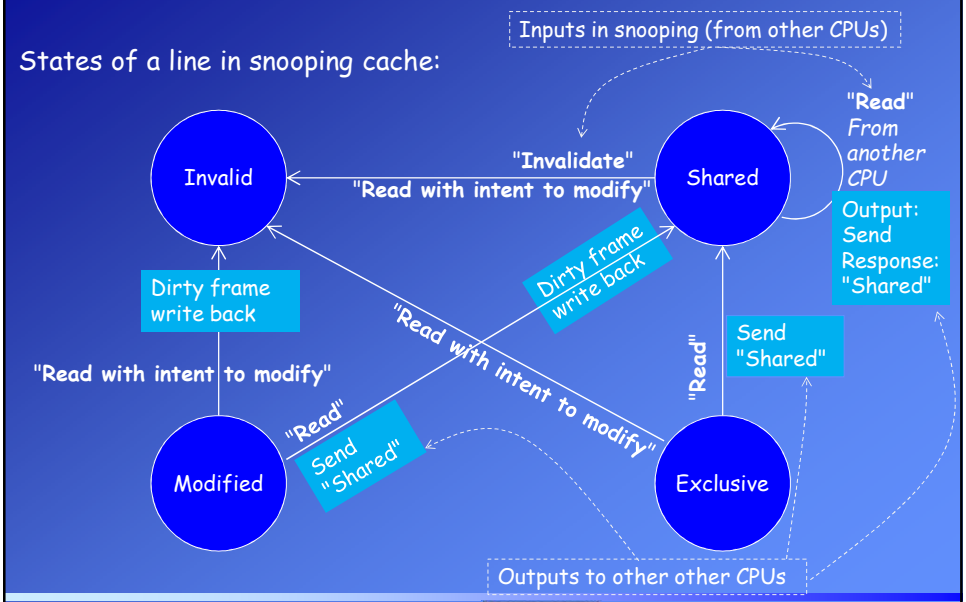
The state of a cache frame can change because of two reasons:

- Due to local processor activity (i.e., cache access)
- Due to bus activity in snooping (activity caused by another processor)

State transition diagram for locally initiated accesses:



State transition diagram for remotely initiated accesses (in snooping):



The MESI Protocol (cont'd)

Operation of the protocol:

Read Miss (Invalid state):

- The processor starts to fetch the frame from main memory.
- The CPU signals other cache controllers to snoop the operation.
- There are four possible outcomes:

A. If **another cache** has an **unmodified** (clean) **exclusive** copy, it indicates that it shares this data.

The responding processor then transitions the state of its copy from exclusive to shared state.

The initiating CPU reads the frame from memory and transitions the cache frame from invalid to shared.

B. If **other caches** have **unmodified** (clean) **shared** copies, they indicate that they share this data.

The responding cache frames stay in the shared state.

The initiating CPU reads the frame and transitions the cache frame from invalid to shared.

Read Miss (Invalid state) cont'd:

- Possible responses (cont'd):

C. If **another cache** has a **modified** (dirty) copy, it blocks the memory read operation and provides the requested frame.

This data is also written to main memory.

There are different implementations. The requesting CPU can read the data from the responding CPU or from main memory after the memory has been updated.

The responding CPU changes its line from modified to shared.

The initiating CPU transitions the cache frame from invalid to shared.

D. If **no other cache** has a copy of the requested frame, then no signals are returned.

The initiating CPU reads the frame from memory and transitions the cache frame from invalid to exclusive.

Read Hit:

- The CPU simply reads the required data from the cache.
- The cache frame remains in the same (current) state: modified, shared, or exclusive.

Write Miss (Invalid state):

- The processor starts to fetch the frame from main memory.
- The CPU issues the signal *read-with-intent-to-modify* on the bus.
- There are two possible scenarios:

A. If another cache has a modified copy of the frame, it signals the requesting CPU (some words in this frame have been modified).

The requesting CPU terminates the bus operation and waits.

The other CPU writes the modified cache frame back to main memory, and transitions the state of the cache from modified to invalid.

The initiating CPU again issues the signal *read-with-intent-to-modify* on the bus and reads the frame from main memory.

The CPU modifies the word in the frame and transitions the state of the frame to modified.

B. If no other cache has a modified copy of the requested frame, then no signals are returned.

The initiating CPU reads the frame from main memory and modifies it.

If one or more caches have a clean copy of the frame in the shared or exclusive state, each cache invalidates its copy of the frame.

Write Hit:

The CPU attempts to write (modify a variable), and the variable (frame) is in the local cache.

Operations depend on the state of the frame being modified.

Shared:

- The CPU broadcasts the "invalidate" signal on the shared bus.
- Each CPU that has a shared copy of the frame in its cache transitions the state of that frame from "shared" to "invalid".
- The initiating CPU updates the variable and transitions its copy of the frame from "shared" to "modified".

Exclusive:

- The CPU already has the sole (exclusive) copy of the data.
- The CPU updates the variable and transitions its copy of the frame from "exclusive" to "modified".

Modified:

- The CPU already has the sole modified copy of the data.
- The CPU updates the variable. The state remains as "modified".

Example:

In a symmetric multiprocessor (SMP) system with a shared bus, there are two CPUs (CPU1 and CPU2) that have local cache memories.

The system does not have a shared L2 cache.

The cache control units use the set associative mapping technique, where each set contains two frames (two-way set associative).

In write operations, Flagged Write Back (FWB) with Write Allocate (WA) methods are used.

Assume that there is a shared variable X in the system. To provide cache coherence, the snoopy MESI protocol is used.

The following questions can be answered independently.

a) Assume that caches of both CPUs include valid copies of variable X. If the copy of X is in set:1, frame:0 in the cache of CPU1, can we know its location in the cache of CPU2? Why?

Solution:

In a symmetric multiprocessor (SMP) system, CPUs use the same memory space. Therefore, variable X has the same address in spaces of both CPU1 and CPU2.

If it is in set:1, frame:0 in the cache of CPU1, then it must be also in set:1 in the cache of CPU2. However, we cannot know which frame of set 1 it is in.

Example (cont'd):

b) Assume that the frame in the cache of CPU1 containing variable X is in state "exclusive". What is the state of the corresponding frame in the cache of CPU2?

Solution:

In this case, valid copies of variable X are in main memory and in the cache of CPU1. Therefore, the state of the corresponding frame in the cache of CPU2 must be in state "**invalid**".

c) Assume that the frame in the cache of CPU1 containing variable X is in state "modified", and CPU2 wants to write to variable X. List the operations performed by the MESI protocol.

Solution:

If it is in state "modified" in CPU1, then it does not exist (invalid) in CPU2.

- CPU2 (write miss) issues the signal *read-with-intent-to-modify*.
- CPU1 signals the requesting CPU2 "*Main memory is not valid*".
- CPU1 writes the modified cache frame back to main memory and transitions the state of the cache from "modified" to "invalid".
- CPU2 issues the signal *read-with-intent-to-modify* again and reads the frame from main memory.
- CPU2 modifies the word in the frame and transitions the state of the frame to "modified".

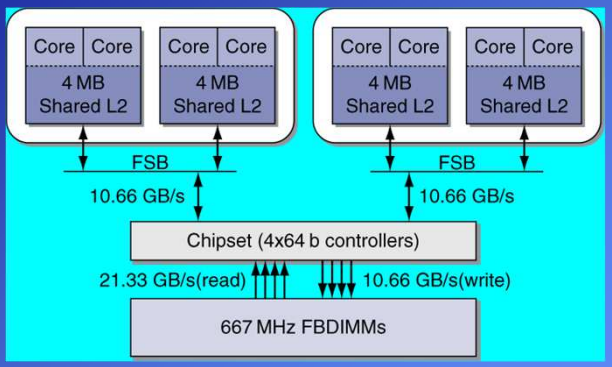
9.5 Challenges of parallel processing

- Limited parallelism in programs. Some processors cannot be used (load imbalance).
- Overhead for communication. High cost of communications between processors.
- Writing parallel programs is difficult.
- Partitioning into independent parts with similar loads: Scheduling and load balancing problem.
- Synchronization: Dependencies, critical sections

9.6 Exemplary Multiprocessor Systems [1]

When more than one processor is implemented on a single chip, the system is called a **multicore chip processor**.

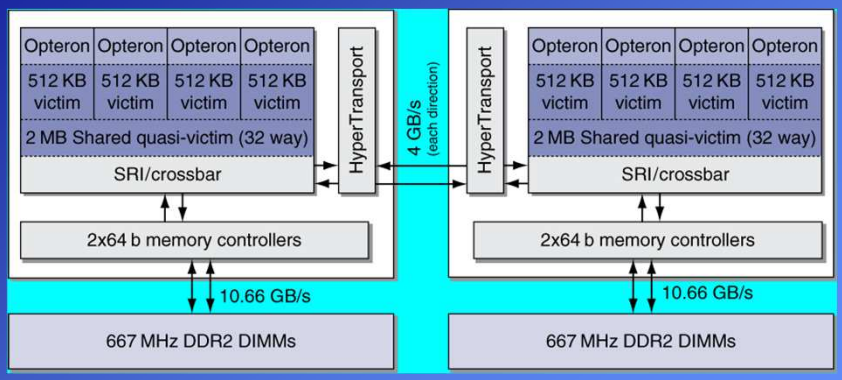
A SMP system with 2 × quad-core Intel Xeon e5345 (Clovertown):



L1 Cache:
128 KB (code)
128 KB (data)

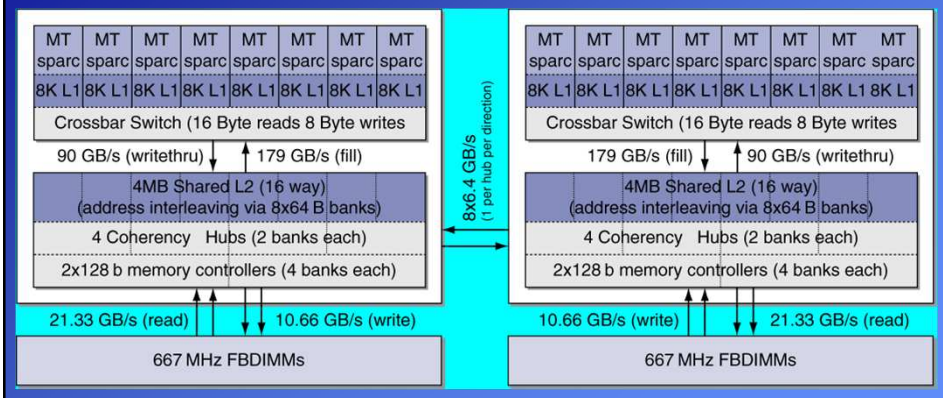
1 Source: Patterson & Hennessy, *Computer Organization and Design, Morgan Kaufmann, Elsevier, 2009.*

2 × quad-core AMD Opteron 2356 (Barcelona)



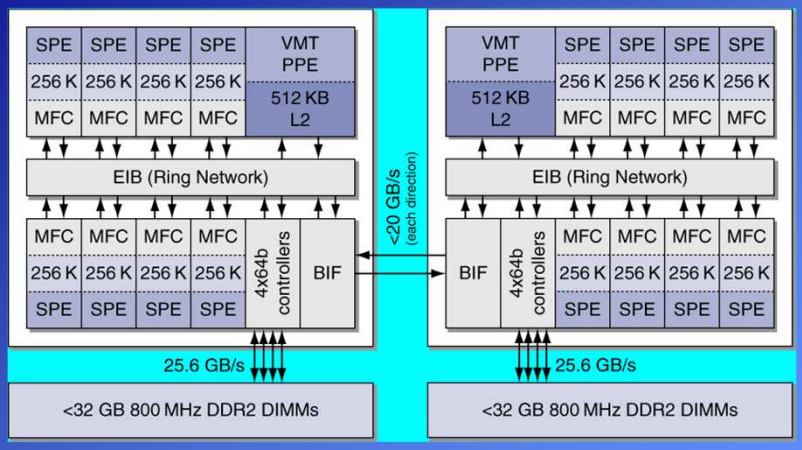
Source: Patterson & Hennessy, *Computer Organization and Design, Morgan Kaufmann, Elsevier, 2009.*

2 × oct-core Sun UltraSPARC T2 5140 (Niagara 2)



Source: Patterson & Hennessy, *Computer Organization and Design, Morgan Kaufmann, Elsevier, 2009.*

2 × oct-core IBM Cell QS20:



Source: Patterson & Hennessy, *Computer Organization and Design, Morgan Kaufmann, Elsevier, 2009.*

The performance wall and search for new solutions *

Computing has evolved because of improvements in semiconductor devices (transistors) and computer architecture (cache memories, pipeline, etc.).

However, these improvements (especially, Moore's Law) are ending.

Designers increase the clock frequency and/or the number of transistors in an integrated circuit (IC) to increase the processing speed of computers.

However, this causes heat/cooling problems (power wall).

Architectural solutions such as pipelining and multicore systems also have their own problems.

However, demands for performance in excess of 1 million trillion floating-point operations per second (1 exaflops) are arising from novel software paradigms to address problems in big data, machine learning, and security.

Many industry experts believe that, by 2020, computing will reach the long-predicted **performance wall**.

Visit the web site of the IEEE Rebooting Computing Initiative to explore the future of computing systems in the architecture, device, and circuit domains.

http://rebootingcomputing.ieee.org/

*Source: T. M. Conte, E. P. DeBenedictis, P. A. Gargini, and E. Track, "Rebooting Computing: The Road Ahead," Computer, vol. 50, no. 1, pp. 20–29, Jan. 2017.