UNIT-5

Multiprocessor Architecture: Taxonomy of Parallel Architectures

1. PARALLEL PROCESSING ARCHITECTURES

Parallel computing architectures breaks the job into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

Multiprocessor: A Computer system with atleast two processors.

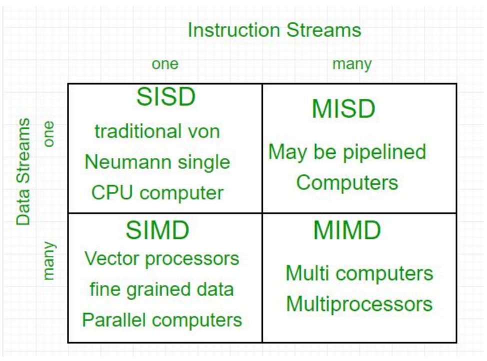
Job – level parallelism (or) process – level parallelism: Utilizing multiple processors by running independent programs simultaneously.

Parallel Processing Program: A single program that runs on multiple processor simultaneously.

Multicore microprocessor: A microprocessor containing multiple processors("Cores") in a single integrated circuit. Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. The crux of parallel processing are the CPUs.

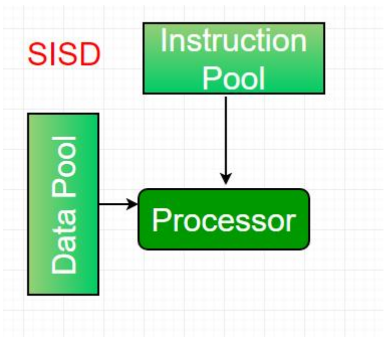
Parallelism in computer architecture is explained used Flynn"s taxonomy. This classification is based on the number of instruction and data streams used in the architecture. The machine structure is explained using streams which are sequence of items. The four categories in Flynn's taxonomy based on the number of instruction streams and data streams are the following:

- (SISD)single instruction,single data
- (MISD)multiple instruction,single data
- (SIMD)single instruction,multiple data
- (MIMD) multiple instruction, multiple data


Instruction Streams
one
many
Data Streams
one
many
SISD
traditional von Neumann single CPU computer
MISD
May be pipelined Computers
SIMD
Vector processors fine grained data Parallel computers
MIMD
Multi computers Multiprocessors

Single-instruction, single-data (SISD) systems –

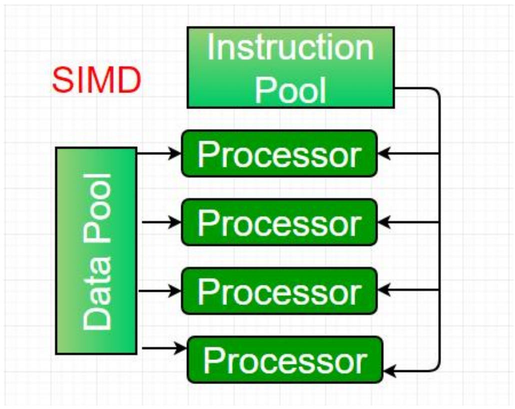
An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.



The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, workstations.

Single-instruction, multiple-data (SIMD) systems –

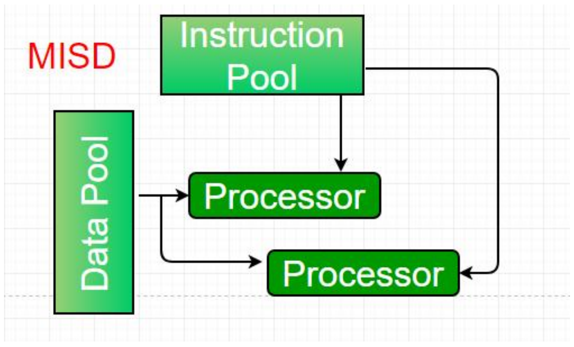
An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.



Dominant representative SIMD systems is Cray’s vector processing machine.

Multiple-instruction, single-data (MISD) systems –

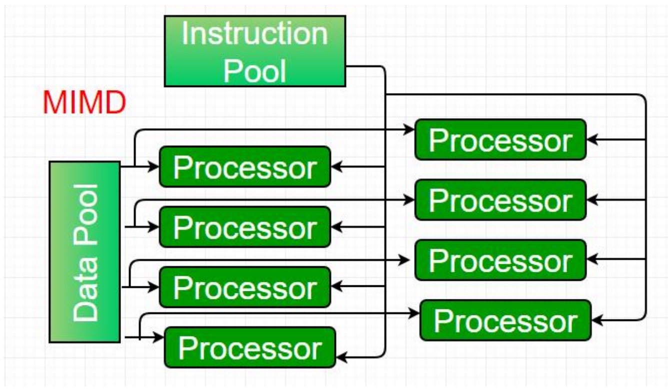
An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .



The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.

Multiple-instruction, multiple-data (MIMD) systems –

An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.



MIMD machines are broadly categorized into shared-memoryMIMD and distributed-memory MIMD based on the way PEs are coupled to the main memory.

2. Centralized Shared-Memory Architectures

- The use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor is the key insight that motivates centralized memory multiprocessors.
- These processors were all single-core and often took an entire board, and memory was located on a shared bus.
- With more recent, higher-performance processors, the memory demands have outstripped the capability of reasonable buses, and recent microprocessors directly connect memory to a single chip, which is sometimes called a backside or memory bus to distinguish it from the bus used to connect to I/O.
- Accessing a chip's local memory whether for an I/O operation or for an access from another chip requires going through the chip that "owns" that memory.
- Thus access to memory is asymmetric: faster to the local memory and slower to the remote memory.
- In a multicore that memory is shared among all the cores on a single chip, but the asymmetric access to the memory of one multicore from the memory of another usually remains.
- Symmetric shared-memory machines usually support the caching of both shared and private data.
- Private data are used by a single processor, while shared data are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data.
- When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Because no other processor uses the data, the program behavior is identical to that in a uniprocessor.
- When shared data are cached, the shared value may be replicated in multiple caches.
- In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously.

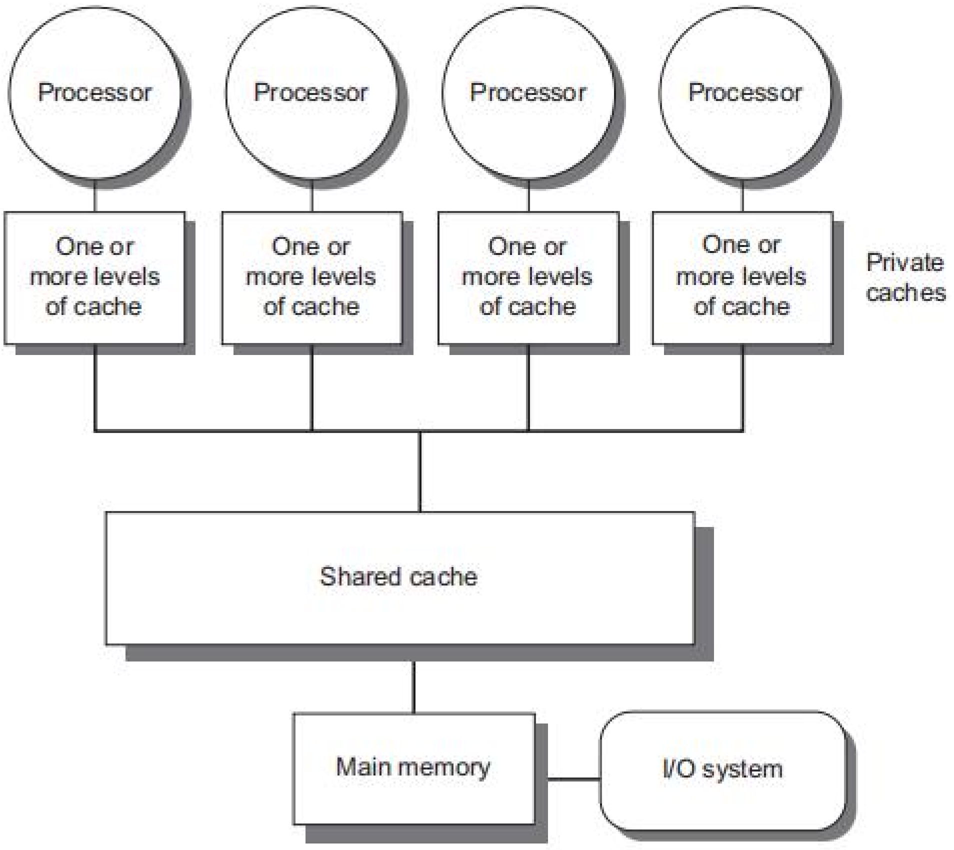

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.

Multiprocessor Cache Coherence

- Unfortunately, caching shared data introduces a new problem. Because the view of memory held by two different processors is through their individual caches, the processors could end up seeing different values for the same memory location.
- This difficulty is generally referred to as the cache coherence problem. Notice that the coherence problem exists because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual caches, which are private to each processor core
- A memory system is coherent if any read of a data item returns the most recently written value of that data item.

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

Figure 5.3 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 write-through 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!

- A memory system is coherent if

1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

- The first property simply preserves program order—we expect this property to be true even in uniprocessors.
- The second property defines the notion of what it means to have a coherent view of memory: if a processor could continuously read an old data value, we would clearly say that memory was incoherent.
- The need for write serialization is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X.
- Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processors could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely.
- The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the same order; this property is called write serialization.
- Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important.
- Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Basic Schemes for Enforcing Coherence

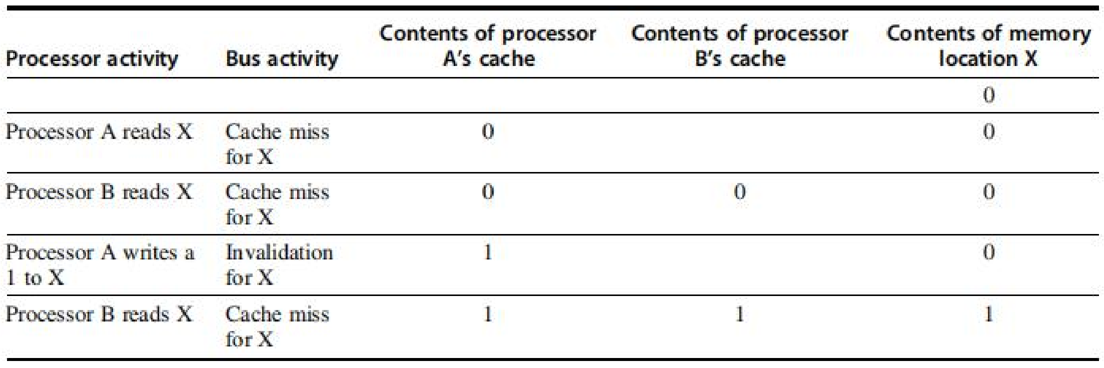
- The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution.
- Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches.
- In a coherent multiprocessor, the caches provide both migration and replication of shared data items.
- Coherent caches provide migration because a data item can be moved to a local cache and used there in a transparent fashion.
- This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
- Because the caches make a copy of the data item in the local cache, coherent caches also provide replication for shared data that are being read simultaneously.
- Replication reduces both latency of access and contention for a read shared data item.
- Supporting this migration and replication is critical to performance in accessing shared data.
- Thus, rather than trying to solve the problem by avoiding it in software, multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.
- The protocols to maintain coherence for multiple processors are called cache coherence protocols.
- Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block.
- The state of any cache block is kept using status bits associated with the block, similar to the valid and dirty bits kept in a uniprocessor cache.
- There are two classes of protocols in use, each of which uses different techniques to track the sharing status:
- **Directory based**—The sharing status of a particular block of physical memory is kept in one location, called the directory. There are two very different types of directory-based cache coherence. In an SMP, we can use one centralized directory, associated with the memory or some other single serialization point, such as the outermost cache in a multicore. In a DSM, it makes no sense to have a single directory because that would create a single point of contention and make it difficult to scale to many multicore chips given the memory demands of multicores with eight or more cores.
- **Snooping**—Rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of the block. In an SMP, the caches are typically all accessible via some broadcast medium (e.g., a bus connects the per-core caches to the shared cache or memory), and all cache controllers monitor or snoop on the medium to determine whether they have a copy of a block that is requested on a bus or

switch access. Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore.

- Snooping protocols became popular with multiprocessors using microprocessors (single-core) and caches attached to a single shared memory by a bus.
- The bus provided a convenient broadcast medium to implement the snooping protocols.

Snooping Coherence Protocols

- There are two ways to maintain the coherence requirement described in the prior section.
- One method is to ensure that a processor has exclusive access to a data item before writing that item.
- This style of protocol is called a write invalidate protocol because it invalidates other copies on a write. It is by far the most common protocol.
- Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.
- Shows an example of an invalidation protocol with write-back caches in action. To see how this protocol ensures coherence, consider a write followed by a read by another processor: because the write requires exclusive access, any copy held by the reading processor must be invalidated (thus the protocol name).
- Therefore when the read occurs, it misses in the cache and is forced to fetch a new copy of the data.
- For a write, we require that the writing processor has exclusive access, preventing any other processor from being able to write simultaneously.
- If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated.
- For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore this protocol enforces write serialization.

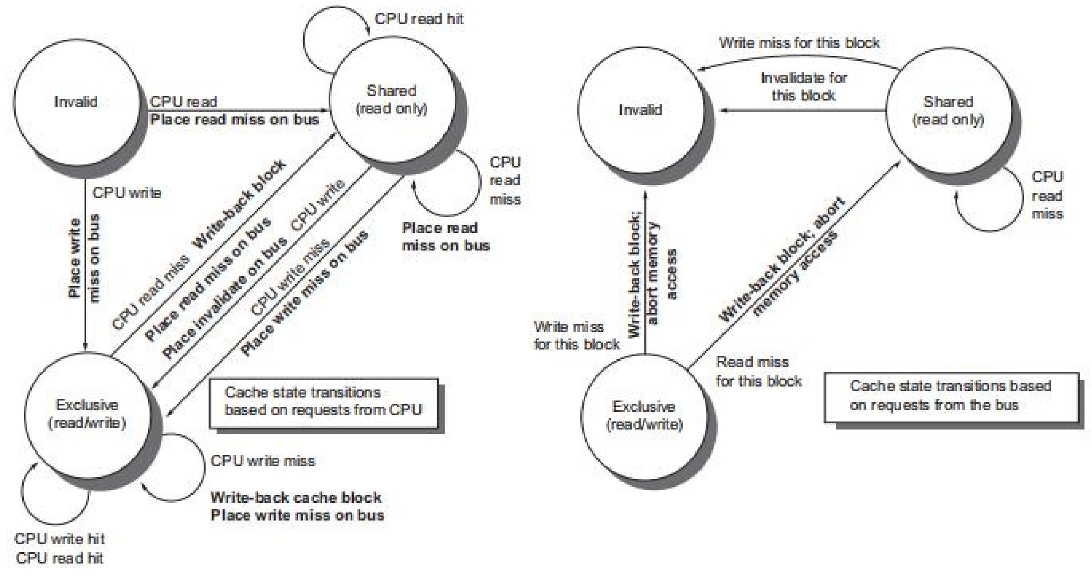


Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for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

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, processor A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional status bit indicating ownership of a block. The ownership bit indicates that a block may be shared for reads, but only the owning processor can write the block, and that processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 caches for each core. It is this observation that led some designers to opt for a directory protocol within the multicore.

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 local 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 local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because 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 shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, 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 shared block; 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 in the local cache.

Figure 5.5 The cache coherence mechanism receives requests from both the core's processor and the shared bus and responds to these based on the type of request, whether it hits or misses in the local cache, and the state of the local cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read, misses, write misses, or invalidates snooped from the bus, an action is required *only* if the read or write addresses match a block in the local cache and the block is valid.

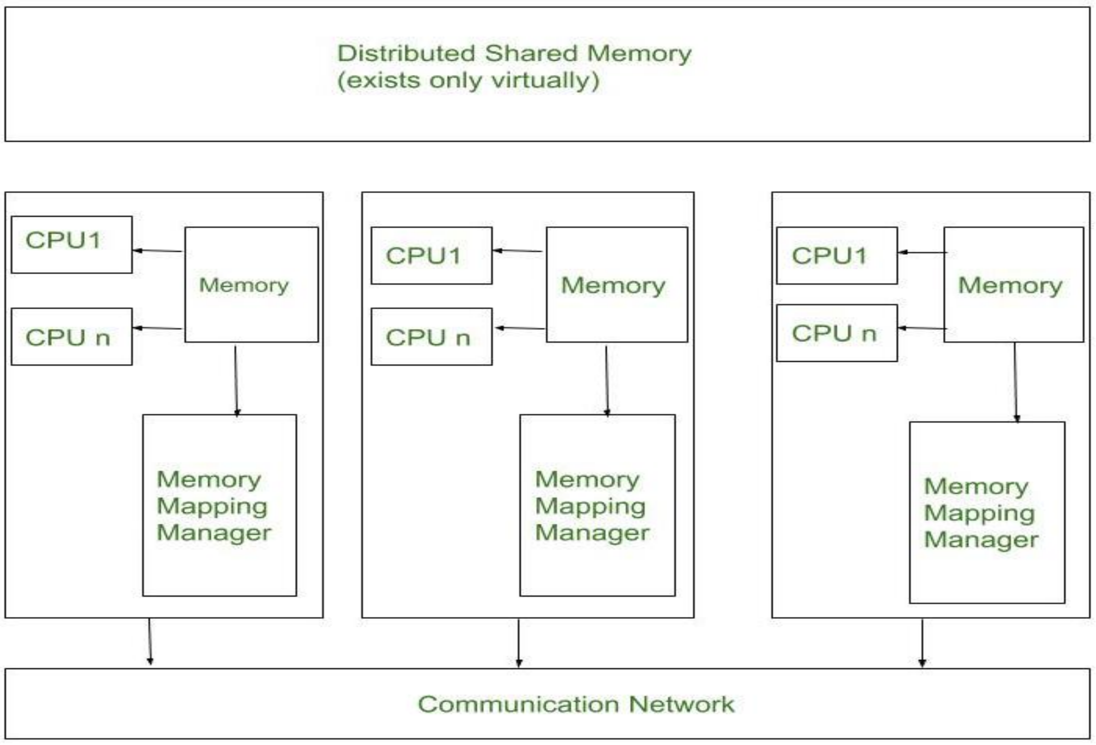


A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the local processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the private cache, not to a specific address in the cache. Thus a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the local cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all private caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol

assumes that memory (or a shared cache) provides data on a read miss for a block that is clean in all local caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss. In a multicore chip, the shared cache (usually L3, but sometimes L2) acts as the equivalent of memory, and the bus is the bus between the private caches of each core and the shared cache, which in turn interfaces to the memory

Architecture of Distributed Shared Memory(DSM)

- Distributed Shared Memory (DSM) implements the distributed systems shared memory model in a distributed system, that hasn't any physically shared memory.
- Shared model provides a virtual address area shared between any or all nodes.
- To beat the high forged of communication in distributed system. DSM memo, model provides a virtual address area shared between all nodes. systems move information to the placement of access.
- Information moves between main memory and secondary memory (within a node) and between main recollections of various nodes.
- Every Greek deity object is in hand by a node.
- The initial owner is that the node that created the object. possession will amendment as the object moves from node to node.
- Once a method accesses information within the shared address space, the mapping manager maps shared memory address to physical memory (local or remote).



- DSM permits programs running on separate reasons to share information while not the software engineer having to agitate causation message instead underlying technology can send the messages to stay the DSM consistent between compute.
- DSM permits programs that wont to treat constant laptop to be simply tailored to control on separate reason. Programs access what seems to them to be traditional memory.
- Hence, programs that Pine Tree State DSM square measure sometimes shorter and easier to grasp than programs that use message passing. But, DSM isn't appropriate for all things.

- Client-server systems square measure typically less suited to DSM, however, a server is also wont to assist in providing DSM practicality for information shared between purchasers.

Architecture of Distributed Shared Memory (DSM) :

- Every node consists of 1 or additional CPU's and a memory unit. High-speed communication network is employed for connecting the nodes.
- A straightforward message passing system permits processes on completely different nodes to exchange one another.

Memory mapping manager unit :

- Memory mapping manager routine in every node maps the native memory onto the shared computer storage.
- For mapping operation, the shared memory house is divided into blocks.
- Information caching may be a documented answer to deal with operation latency.
- DMA uses information caching to scale back network latency. the most memory of the individual nodes is employed to cache items of the shared memory house.
- Memory mapping manager of every node reads its native memory as an enormous cache of the shared memory house for its associated processors.
- The bass unit of caching may be a memory block. Systems that support DSM, information moves between secondary memory and main memory also as between main reminiscences of various nodes.

Communication Network Unit :

- Once method access information within the shared address house mapping manager maps the shared memory address to the physical memory.
- The mapped layer of code enforced either within the operating kernel or as a runtime routine.
- Physical memory on every node holds pages of shared virtual–address house. Native pages area unit gift in some node's memory.
- Remote pages in some other node's memory.

Types of Distributed shared memory

On-Chip Memory:

- The data is present in the CPU portion of the chip.
- Memory is directly connected to address lines.
- On-Chip Memory DSM is expensive and complex.

Bus-Based Multiprocessors:

- A set of parallel wires called a bus acts as a connection between CPU and memory.
- Accessing of same memory simultaneously by multiple CPUs is prevented by using some algorithms
- Cache memory is used to reduce network traffic.

Ring-Based Multiprocessors:

- There is no global centralized memory present in Ring-based DSM.
- All nodes are connected via a token passing ring.
- In ring-bases DSM a single address line is divided into the shared area.

Advantages of Distributed shared memory

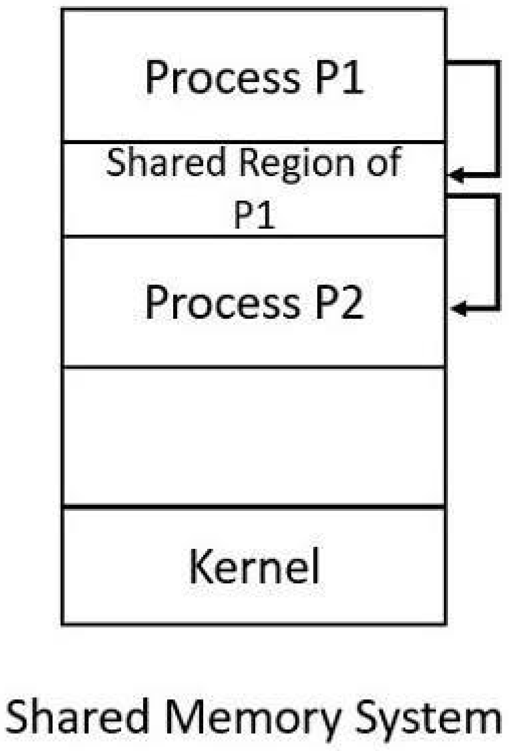
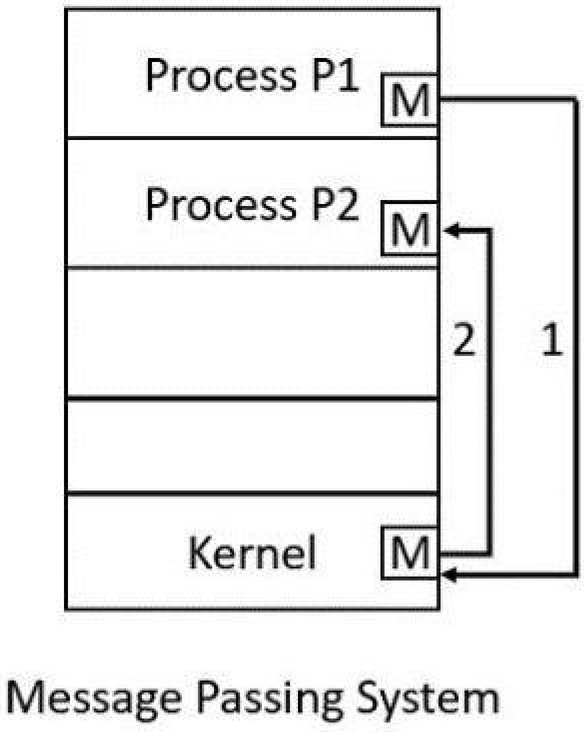
- **Simpler abstraction:** Programmer need not concern about data movement, As the address space is the same it is easier to implement than RPC.
- **Easier portability:** The access protocols used in DSM allow for a natural transition from sequential to distributed systems. DSM programs are portable as they use a common programming interface.
- **locality of data:** Data moved in large blocks i.e. data near to the current memory location that is being fetched, may be needed future so it will be also fetched.
- **on-demand data movement:** It provided by DSM will eliminate the data exchange phase.
- **larger memory space:** It provides large virtual memory space, the total memory size is the sum of the memory size of all the nodes, paging activities are reduced.
- **Better Performance:** DSM improve performance and efficiency by speeding up access to data.
- **Flexible communication environment:** They can join and leave DSM system without affecting the others as there is no need for sender and receiver to existing,
- **process migration simplified:** They all share the address space so one process can easily be moved to a different machine.

Differentiate between shared memory and message passing

Differences

The major differences between shared memory and message passing model −

Shared Memory	Message Passing
It is one of the region for data communication	Mainly the message passing is used for communication.
It is used for communication between single processor and multiprocessor systems where the processes that are to be communicated present on the same machine and they are sharing common address space.	It is used in distributed environments where the communicating processes are present on remote machines which are connected with the help of a network.
The shared memory code that has to be read or write the data that should be written explicitly by the application programmer.	Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the communicating processes.
It is going to provide a maximum speed of computations because the communication is done with the help of shared memory so system calls are used to establish the shared memory.	Message passing is a time consuming process because it is implemented through kernel (system calls).
In shared memory make sure that the processes are not writing to the same location simultaneously.	Message passing is useful for sharing small amounts of data so that conflicts need not occur.
It follows a faster communication strategy when compared to message passing technique.	In message passing the communication is slower when compared to shared memory technique.
Given below is the structure of shared memory system −	Given below is the structure of message passing system −

Shared Memory	Message Passing
 Process P1 Shared Region of P1 Process P2 Kernel 1 2 Shared Memory System	 Process P1 M Process P2 M Kernel M 2 1 Message Passing System

Shared memory system

is the fundamental model of inter process communication. In a shared memory system, in the address space region the cooperating communicate with each other by establishing the shared memory region.Shared memory concept works on fastest inter process communication.If the process wants to initiate the communication and it has some data to share, then establish the shared memory region in its address space.After that, another process wants to communicate and tries to read the shared data, and must attach itself to the initiating process's shared address space.

Message Passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.