Computer Organization and Architecture (EET 2211)

Chapter 17: Parallel Processing

Book Referred

Computer Organization and Architecture: Designing for Performance by William Stallings, 10th Edition, Pearson Ed. Ltd.

Topics of Discussion

Lecture 1

17.1 Multiple Processor Organizations

17.2 Symmetric Multiprocessors

Lecture 2

17.3 Cache Coherence and MESI protocol

17.4 Multithreading and Chip Multiprocessors

Lecture 3

17.5 Clusters

17.6 Non-uniform Memory Access (NUMA)

Learning Objectives

After studying this chapter you should be able to:

- ❖Summarize the types of parallel processor organizations.
- ❖Present an overview of design features of symmetric multiprocessors.
- ❖Understand the issue of cache coherence in a multiple processor system.
- ❖Explain the key features of the MESI Protocols.
- ❖Explain the difference between the implicit and explicit multithreading.
- ❖Summarize the issues of clusters
- ❖Explain the concept of non-uniform memory access.

LECTURE 33

1. Multiple Processor Organizations

Types of parallel processor systems as proposed by Flynn:

1. **Single Instruction, Single Data (SISD) stream**: A single processor executes a single instruction stream to operate on data stored in a single memory. Eg. Uniprocessors

2. **Single Instruction, Multiple Data (SIMD) stream**: A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that instructions are executed on different sets of data by different processors. Eg.: Array and Vector processors

3. **Multiple Instruction, Single Data (MISD) stream**: A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.

4. **Multiple Instruction, Multiple Data (MIMD) stream**: A set of processors simultaneously execute different instruction sequences on different data sets. Eg.: SMPs, Clusters, NUMA systems

- With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation.
- MIMDs can be further subdivided by the means in which the processors communicate.
- In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor.
- In **nonuniform memory access (NUMA)** organization, the memory access time to different regions of memory may differ for a NUMA processor.
- A collection of independent uniprocessors or SMPs may be interconnected to form a **cluster**. Communication among the computers is either via fixed paths or via some network facility.

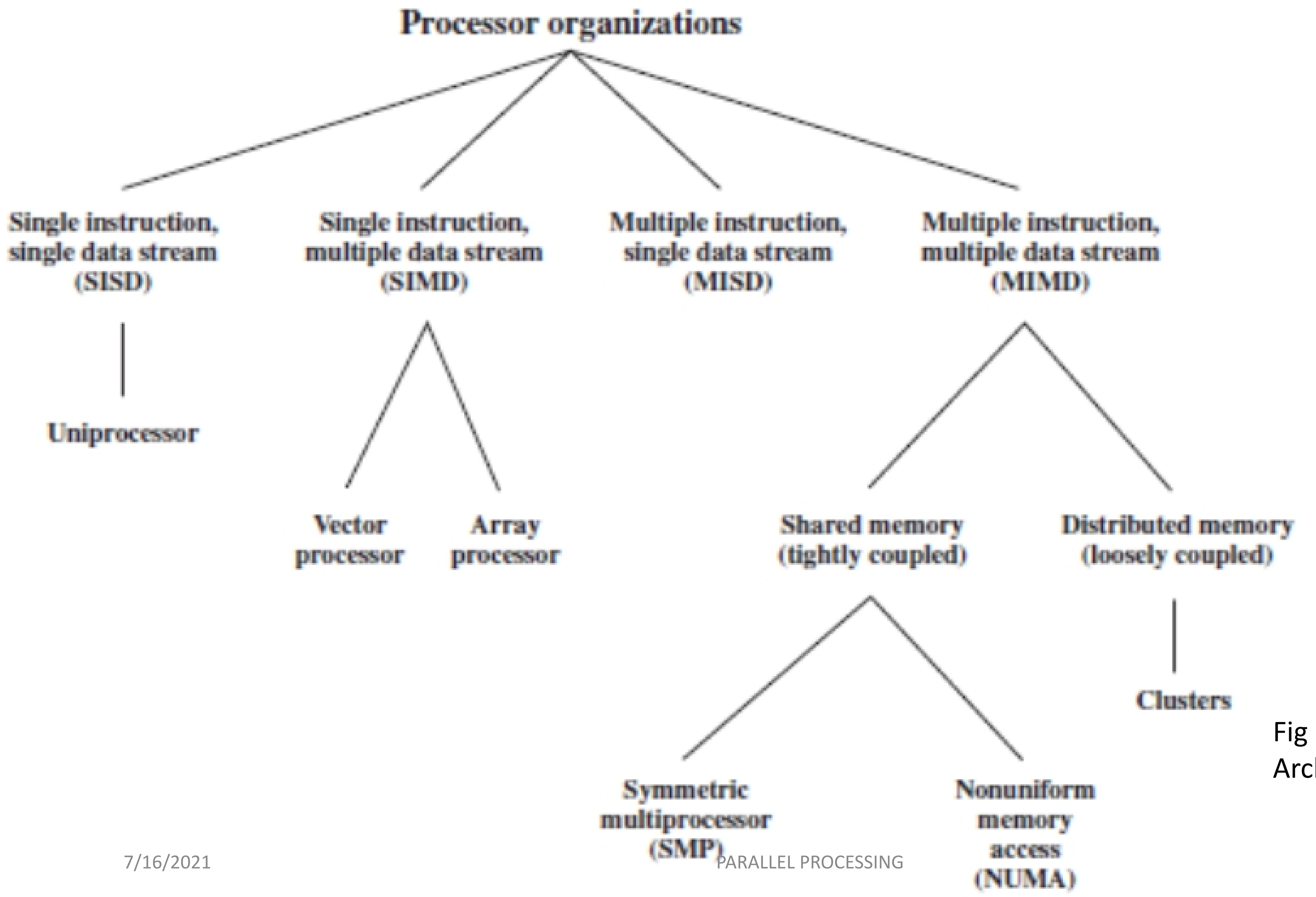
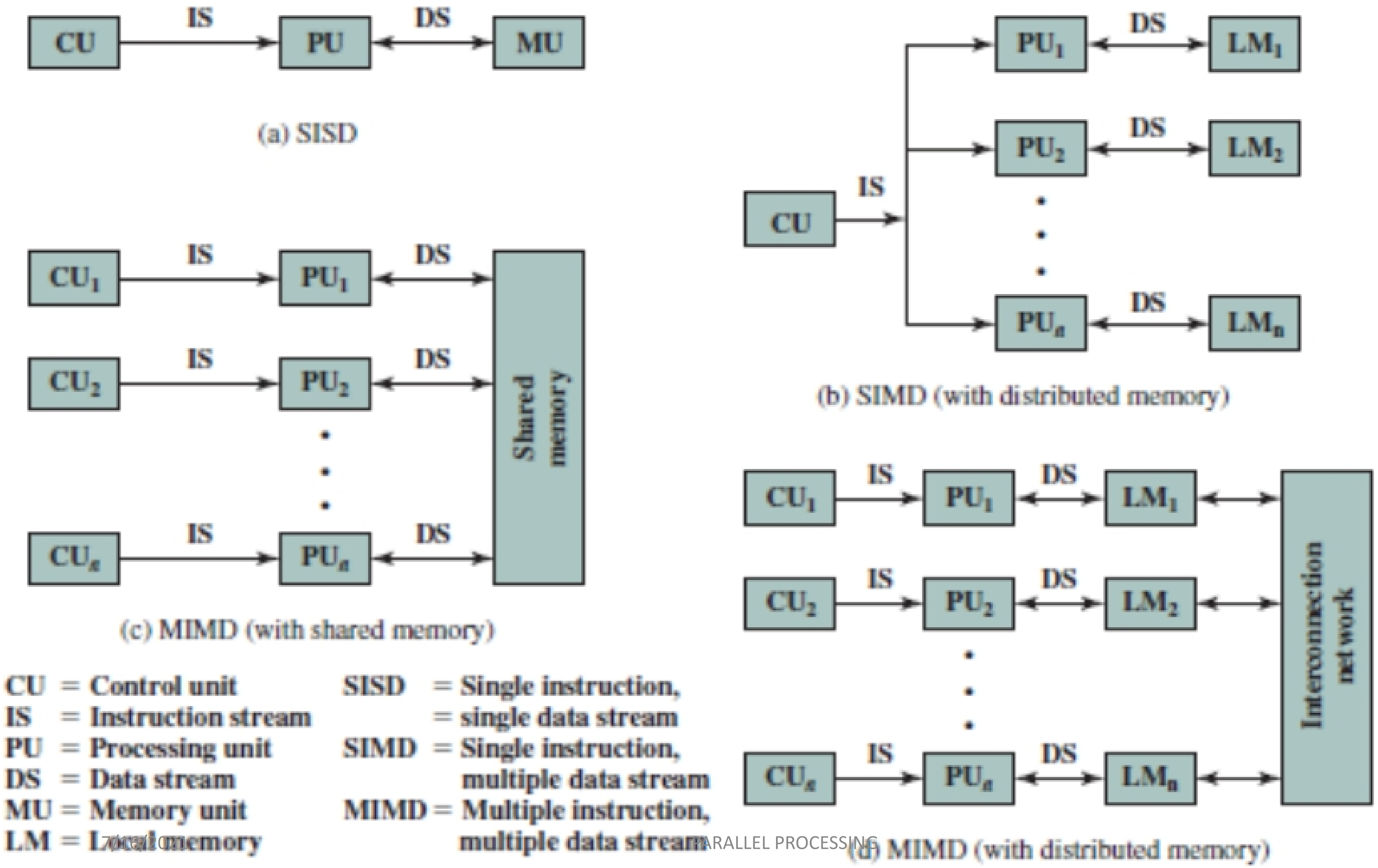

Processor organizations
Single instruction, single data stream (SISD)
Single instruction, multiple data stream (SIMD)
Multiple instruction, single data stream (MISD)
Multiple instruction, multiple data stream (MIMD)
Uniprocessor
Vector processor
Array processor
Shared memory (tightly coupled)
Distributed memory (loosely coupled)
Clusters
Symmetric multiprocessor (SMP)
Nonuniform memory access (NUMA)

Fig 1: Parallel Processor Architecture



CU	= Control unit	SISD	= Single instruction, = single data stream
IS	= Instruction stream		
PU	= Processing unit	SIMD	= Single instruction, multiple data stream
DS	= Data stream		
MU	= Memory unit	MIMD	= Multiple instruction, multiple data stream
LM	= Local memory		

Fig 2: Alternative Computer Organizations

2. Symmetric Multiprocessors (SMP)

- An SMP can be defined as standalone computer system with following characteristics:
  1. There are two or more similar processors of comparable capability.
  2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
  3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
  4. All processors can perform the same functions (hence the term *symmetric*).
  5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

- The operating system of an SMP schedules processes or threads across all of the processors.
- An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:
  - ➢**Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.
  - ➢**Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
  - ➢**Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
  - ➢**Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.
  - ➢The existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

Organization

In general terms, in a multiprocessor system:

- There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache.
- Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism.
- The processors can communicate with each other through memory (messages and status information left in common data areas) or even exchange signals directly.
- The memory is so organized that multiple simultaneous accesses to separate blocks of memory are possible.
- In some configurations, each processor may also have its own private main memory and I/O channels in addition to the shared resources.

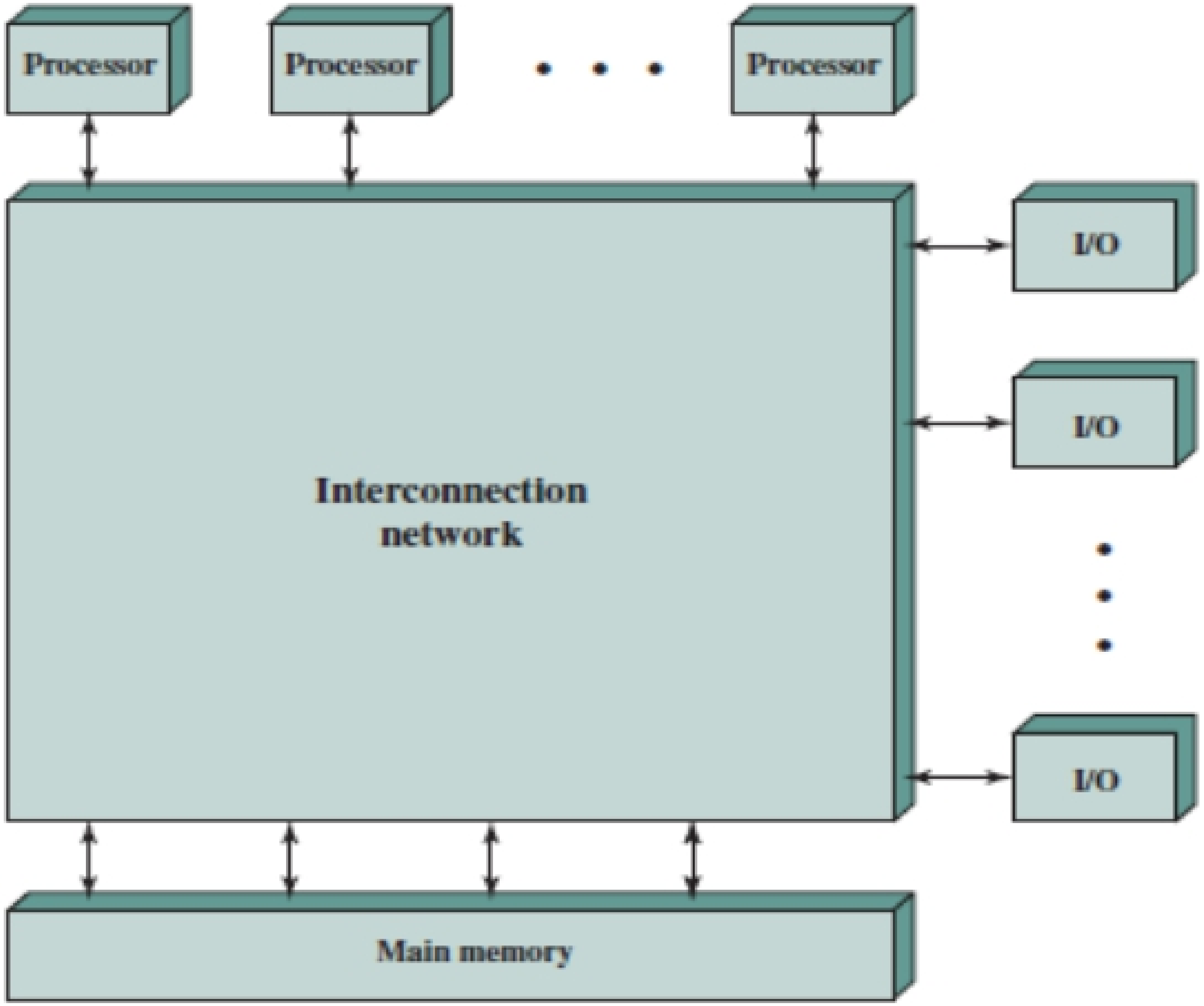


Fig 3: Generic block diagram of Tightly Coupled Multiprocessor

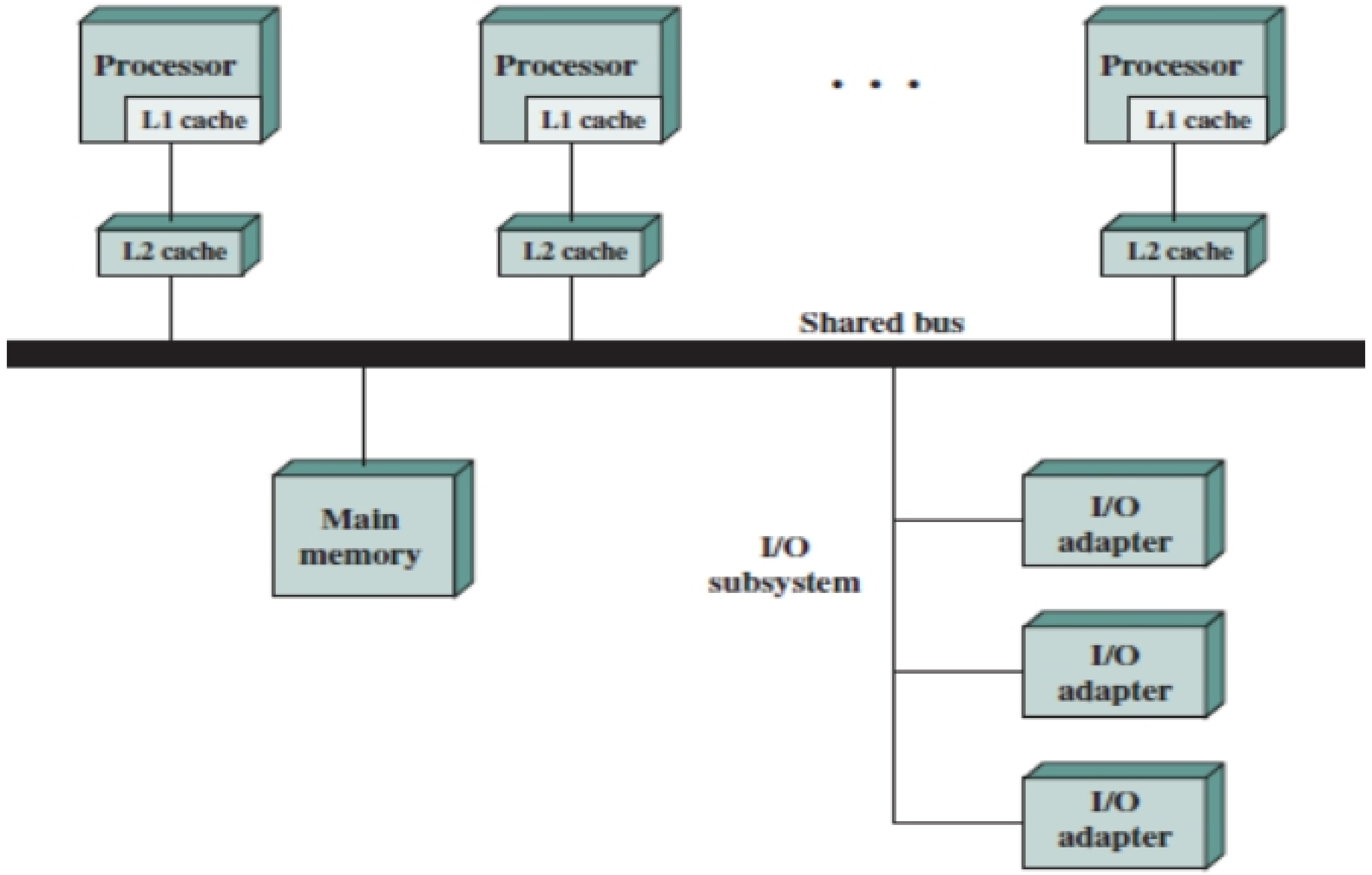


Fig 4: Symmetric Multiprocessor Organization

- The most common organization for personal computers, workstations, and servers is the time-shared bus.
- The time-shared bus is the simplest mechanism for constructing a multiprocessor system.
- The structure and interfaces are basically the same as for a single-processor system that uses a bus interconnection.
- The bus consists of control, address, and data lines.

- The bus organization has several attractive features:

➢ **Simplicity:** This is the simplest approach to multiprocessor organization. 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.

➢ **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

- Performance is the main drawback as All memory references pass through the common bus. Thus, the bus cycle time limits the speed of the system.
- To improve performance, it is desirable to equip each processor with a cache memory, thus reducing the number of bus accesses dramatically.

- Typically, workstation and PC SMPs have two levels of cache, with the L1 cache internal (same chip as the processor) and the L2 cache either internal or external. Some processors now employ a L3 cache as well.
- Because each local cache contains an image of a portion of memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache.
- To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the ***cache coherence*** problem and is typically addressed in hardware rather than by the operating system.

Multiprocessor Operating System Design Considerations

- An **SMP operating system** manages processor and other computer resources so that the user perceives a single operating system controlling system resources, instead it should appear as single-processor multiprogramming system.
- It is the responsibility of the operating system to schedule the execution of multiple jobs or processes and to allocate resources.
- A user may construct applications that use multiple processes or multiple threads within processes without regard to the type of system available.
- A multiprocessor operating system must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors.

Key Design Issues of MP-OS

1. **Simultaneous Concurrent processes**: OS routines need to be reentrant to allow several processors to execute the same IS code simultaneously. With multiple processors executing the same or different parts of the OS, OS tables and management structures must be managed properly to avoid deadlock or invalid operations.
2. **Scheduling**: Any processor may perform scheduling, so conflicts must be avoided. The scheduler must assign ready processes to available processors.
3. **Synchronization**: With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective enforcement of mutual exclusion and event ordering.
4. **Memory Management**: must deal with all of the issues found on uniprocessor machines. In addition, the operating system needs to exploit the available hardware parallelism, such as multiported memories, to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement.
5. **Reliability and Fault Tolerance**: The operating system should provide graceful degradation in the face of processor failure. The scheduler and other portions of the operating system must recognize the loss of a processor and restructure management tables accordingly.

LECTURE 34

3. Cache Coherence and MESI Protocol

- **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.
- Two common write policies or writing to the memory:

1. **Write back:** Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is evicted from the cache.
2. **Write through:** All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

- A write-back policy can result in inconsistency. If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequent reads to that invalid line produce invalid results.
- Even with the write-through policy, inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update.
- For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and write, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time.
- Cache coherence approaches have generally been divided into software and hardware approaches. Some implementations adopt a strategy that involves both software and hardware elements.

Software Solutions

- Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem.
- Software approaches are attractive because the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software.
- Compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization.

Compiler-based Coherence Mechanisms

1. Perform an analysis on the code to determine which data items may become unsafe for caching, and
2. They mark those items accordingly.
3. The operating system or hardware then prevents non-cacheable items from being cached.

The simplest approach

- Prevent any shared data variables from being cached.
- This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods.
- It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue.

Efficient Approaches

- Analyze the code to determine safe periods for shared variables.
- The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods.

Hardware Solutions

- Cache coherence protocols.
- These solutions provide dynamic recognition at run time of potential inconsistency conditions.
- Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach.
- These approaches are transparent to the programmer and the compiler, reducing the software development burden.
- Differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms.
- Two Broad categories: Directory Protocols & Snoopy Protocols

Directory Protocols

- Collect and maintain information about where copies of lines reside.
- Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory.
- The directory contains global state information about the contents of the various local caches. When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches.
- It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of a line must be reported to the central controller.

Operation:

1. Typically, the controller maintains information about which processors have a copy of which lines.
2. Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller.
3. Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy.
4. After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor.
5. When another processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller.
6. The controller then issues a command to the processor holding that line that requires the processor to do a write back to main memory.
7. The line may now be shared for reading by the original processor and the requesting processor.

Drawbacks

- Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller.
- However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

Snoopy Protocols

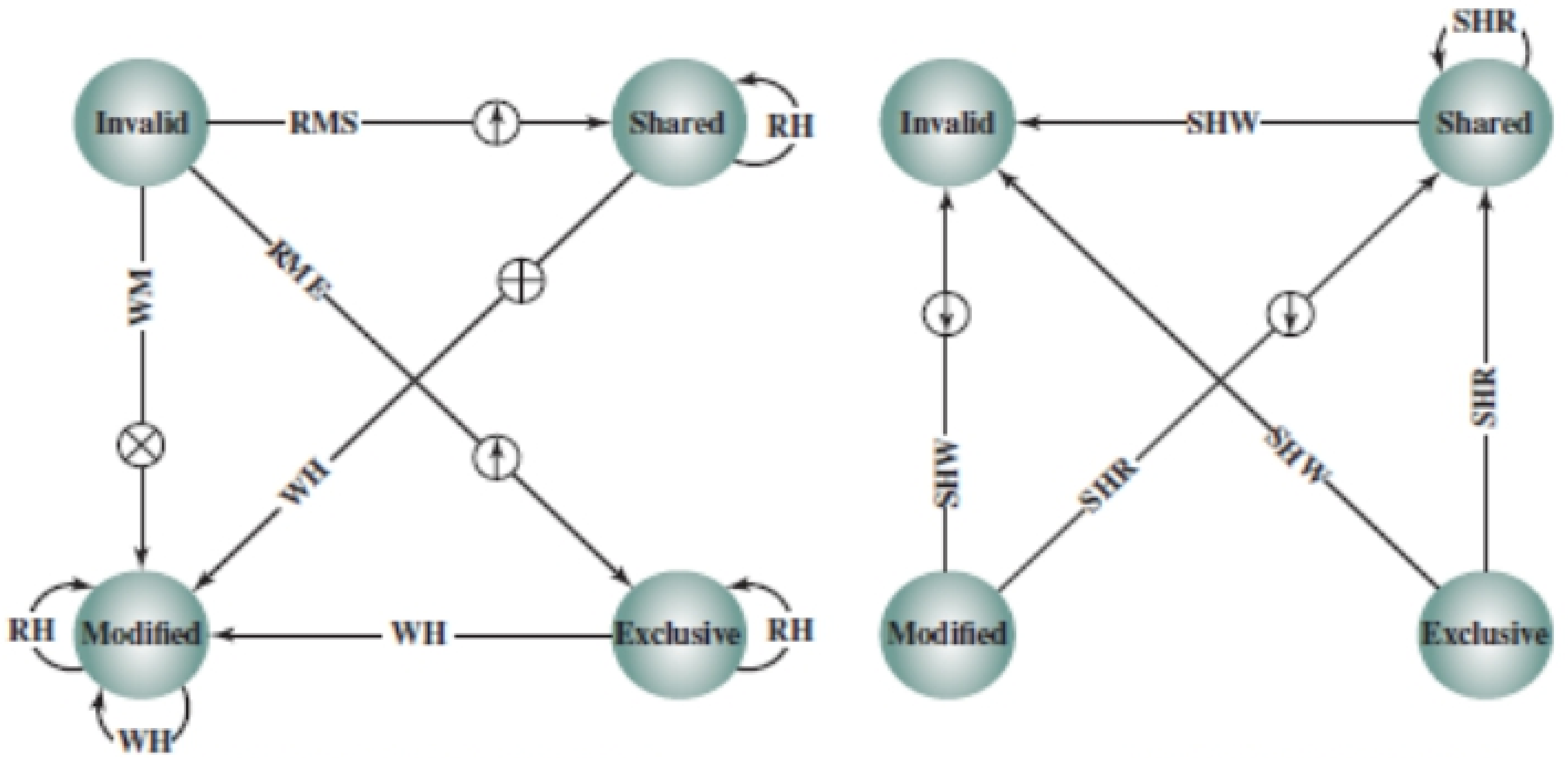
- Distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor.
- A cache must recognize when a line that it holds is shared with other caches.
- When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism.
- Each cache controller is able to “snoop” on the network to observe these broadcasted notifications, and react accordingly.
- Ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping.
- However, because one of the objectives of the use of local caches is to avoid bus accesses, care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches.

- Two basic approaches: write invalidate and write update (or write broadcast).
- With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.
- With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make cheap local writes until some other processor requires the same line.
  - Most widely used in commercial multiprocessor systems, such as the x86 architecture. It marks the state of every cache line (using two extra bits in the cache tag) as modified, exclusive, shared, or invalid.
  - For this reason, the write-invalidate protocol is called MESI.

The MESI Protocol

- To provide cache consistency on an SMP, the data cache often supports a protocol known as MESI.
- For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states, at any given time:

1. **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.
2. **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.
3. **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.
4. **Invalid:** The line in the cache does not contain valid data.

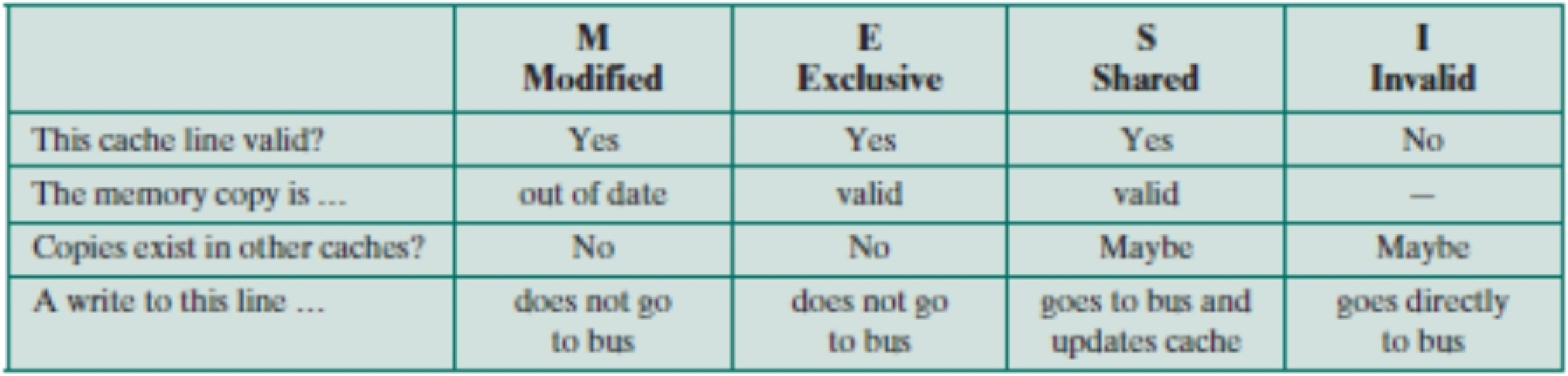


(a) Line in cache at initiating processor

(b) Line in snooping cache

RH	– Read hit	↓	Dirty line copyback
RMS	– Read miss, shared	⊕	Invalidate transaction
RME	– Read miss, exclusive	⊗	Read-with-intent-to-modify
WH	– Write hit	↑	Cache line fill
WM	– Write miss		
SHR	– Snoop hit on read		
SHW	– Snoop hit on write or read-with-intent-to-modify		

Fig 5: MESI State Transition Diagram



	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is …	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line …	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

Fig 6: MESI Cache Line States

Few Terms:

✓ ***Read Miss*** When a read miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. The processor inserts a signal on the bus that alerts all other processor/cache units to snoop the transaction. There are a number of possible outcomes:

■■ If one other cache has a clean (unmodified since read from memory) copy of the line in the exclusive state, it returns a signal indicating that it shares this line. The responding processor then transitions the state of its copy from exclusive to shared, and the initiating processor reads the line from main memory and transitions the line in its cache from invalid to shared.

■■ If one or more caches have a clean copy of the line in the shared state, each of them signals that it shares the line. The initiating processor reads the line and transitions the line in its cache from invalid to shared.

■■ If one other cache has a modified copy of the line, then that cache blocks the memory read and provides the line to the requesting cache over the shared bus. The responding cache then changes its line from modified to shared. The line sent to the requesting cache is also received and processed by the memory controller, which stores the block in memory.

■■ If no other cache has a copy of the line (clean or modified), then no signals are returned. The initiating processor reads the line and transitions the line in its cache from invalid to exclusive.

✓ **Write Miss** - When a write miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. For this purpose, the processor issues a signal on the bus that means *read-with-intent-to-modify* (RWITM). When the line is loaded, it is immediately marked modified. With respect to other caches, two possible scenarios precede the loading of the line of data.

1. First, some other cache may have a modified copy of this line (state = modify). In this case, the alerted processor signals the initiating processor that another processor has a modified copy of the line. The initiating processor surrenders the bus and waits. The other processor gains access to the bus, writes the modified cache line back to main memory, and transitions the state of the cache line to invalid (because the initiating processor is going to modify this line). Subsequently, the initiating processor will again issue a signal to the bus of RWITM and then read the line from main memory, modify the line in the cache, and mark the line in the modified state.
2. The second scenario is that no other cache has a modified copy of the requested line. In this case, no signal is returned, and the initiating processor proceeds to read in the line and modify it. Meanwhile, if one or more caches have a clean copy of the line in the shared state, each cache invalidates its copy of the line, and if one cache has a clean copy of the line in the exclusive state, it invalidates its copy of the line.

- ✓ ***Read Hit*** **-** When a read hit occurs on a line currently in the local cache, the processor simply reads the required item. There is no state change: The state remains modified, shared, or exclusive.
- ✓ ***Write Hit*** - When a write hit occurs on a line currently in the local cache, the effect depends on the current state of that line in the local cache:

■ **Shared:** Before performing the update, the processor must gain exclusive ownership of the line. The processor signals its intent on the bus. Each processor that has a shared copy of the line in its cache transitions the sector from shared to invalid. The initiating processor then performs the update and transitions its copy of the line from shared to modified.

■ **Exclusive:** The processor already has exclusive control of this line, and so it simply performs the update and transitions its copy of the line from exclusive to modified.

■ **Modified:** The processor already has exclusive control of this line and has the line marked as modified, and so it simply performs the update.

L1-L2 Cache Consistency

- Cache coherency protocols are generally applied to L2 caches, since they are connected to same bus, and hence L1 caches cannot participate.
- Strategy: Extend MESI protocol to L1 caches with each line in L1 cache including bits to indicate the state.
- Objectives: for any line that is present in both an L2 cache and its corresponding L1 cache, the L1 line state should track the state of the L2 line. – By adopting write-through policy in L1 cache with write-through to L2 and not the memory.
- The L1 write-through policy forces any modification to an L1 line out to the L2 cache and therefore makes it visible to other L2 caches.
- The use of the L1 write-through policy requires that the L1 content must be a subset of the L2 content.
- This in turn suggests that the associativity of the L2 cache should be equal to or greater than that of the L1 associativity.

4. Multithreading and Chip Multiprocessors

- Performance of a processor – rate at which it executes instructions
- Given by: MIPS rate = processor clock frequency(MHz) x Instructions/cycle
- IPC has increased due to use of pipelined and multiple pipelined architectures and use of ever more complex mechanisms – reached a limit due to complexity and power consumption concerns.
- Alternative approach – **Multithreading** – the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel.

Some Key Definitions

❑ **Process**: An instance of a program running on a computer and has 2 key characteristics:

- **Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.
- **Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

Some Key Definitions (contd.)

- ❑**Process switch**: An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second.
- ❑**Thread**: A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.
- ❑**Thread switch**: The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

- A thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch.

Some Key Definitions (contd.)

- ❑ **User-level threads** are visible to the application program
- ❑ **Kernel-level threads** are visible only to the operating system.
- Both of these may be referred to as explicit threads, defined in software.
- ❑ **Explicit multithreading**: The concurrent execution of instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. All of the commercial processors and most of the experimental processors so far have used explicit multithreading.
- ❑ **Implicit multithreading**: The concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware.

Approaches to Explicit Multithreading

- At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently.
- The designs differ in the amount and type of additional hardware used to support concurrent thread execution.
- In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques.
- Greatly improved performance can be achieved by combining thread-level parallelism and instruction level parallelism.
- There are 4 principal approaches.

- **Interleaved multithreading:** Or **fine-grained multithreading**. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.
- **Blocked multithreading**: Or **coarse-grained multithreading**. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.
- **Simultaneous multithreading (SMT)**: Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.
- **Chip multiprocessing**: Multiple cores are implemented on a single chip and each core handles separate threads. The advantage – the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design.

- For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better utilization of the processor’s execution resources and avoids a large penalty due to cache misses and other latency events.
- The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources.
- Chip multiprocessing also enables simultaneous execution of instructions from different threads.

Multithreading approaches

■ **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading. Refer Fig 7(a).

■ **Interleaved multithreaded scalar:** This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle, the pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles. Refer Fig 7(b).

■ **Blocked multithreaded scalar:** In this case, a single thread is executed until a latency event occurs that would stop the pipeline, at which time the processor switches to another thread. Refer Fig 7(c).

■ **Superscalar:** This is the basic superscalar approach with no multithreading. Until relatively recently, this was the most powerful approach to providing parallelism within a processor. Note that during some cycles, not all of the available issue slots are used. During these cycles, less than the maximum number of instructions is issued; this is referred to as *horizontal loss*. During other instruction cycles, no issue slots are used; these are cycles when no instructions can be issued; this is referred to as *vertical loss*. Refer Fig 7(d).

- In the case of interleaved multithreading, it is assumed that there are no control or data dependencies between threads, which simplifies the pipeline design and therefore should allow a thread switch with no delay.
- However, depending on the specific design and implementation, block multithreading may require a clock cycle to perform a thread switch. This is true if a fetched instruction triggers the thread switch and must be discarded from the pipeline.
- Although interleaved multithreading appears to offer better processor utilization than blocked multithreading, it does so at the sacrifice of single-thread performance. The multiple threads compete for cache resources, which raises the probability of a cache miss for a given thread.

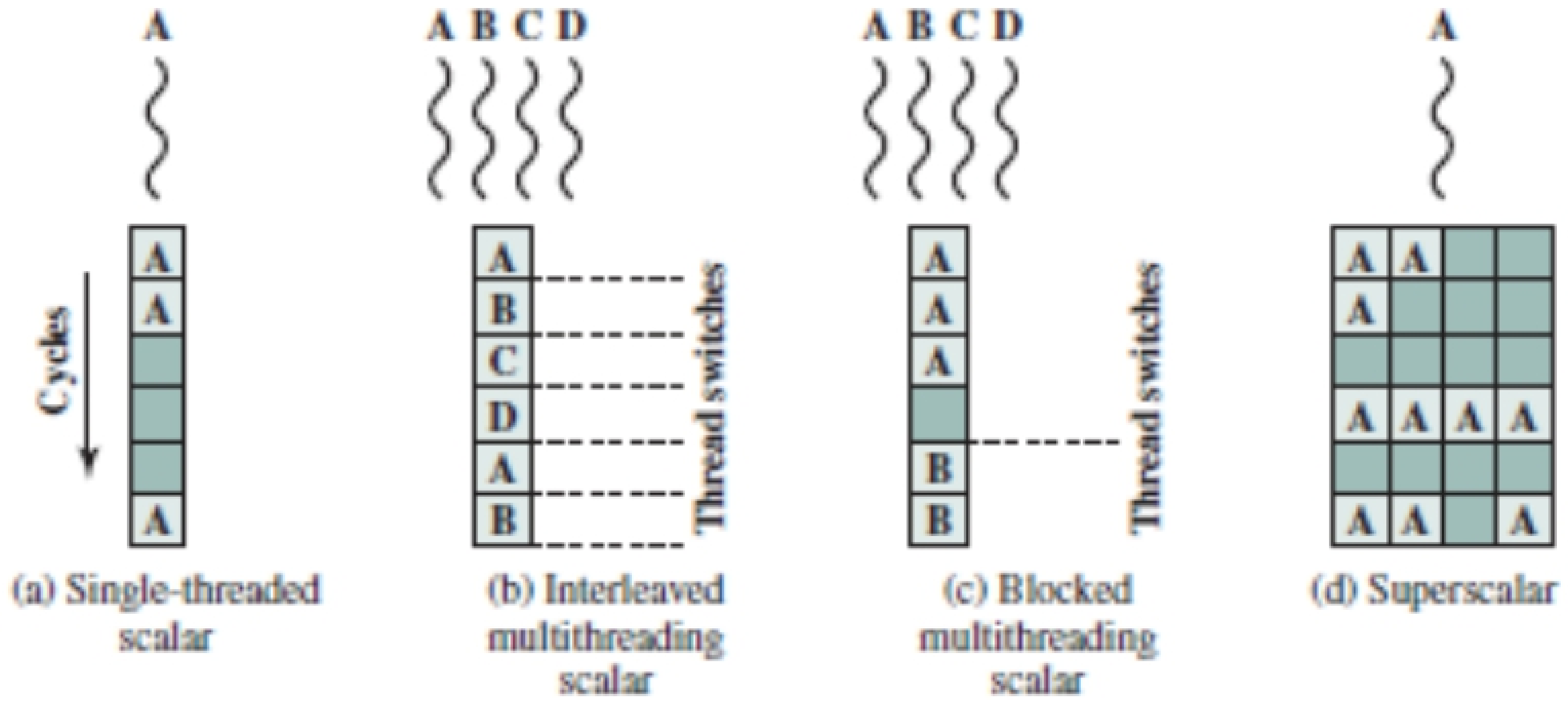


Fig 7 (a-d): Multiple Thread Execution Approaches

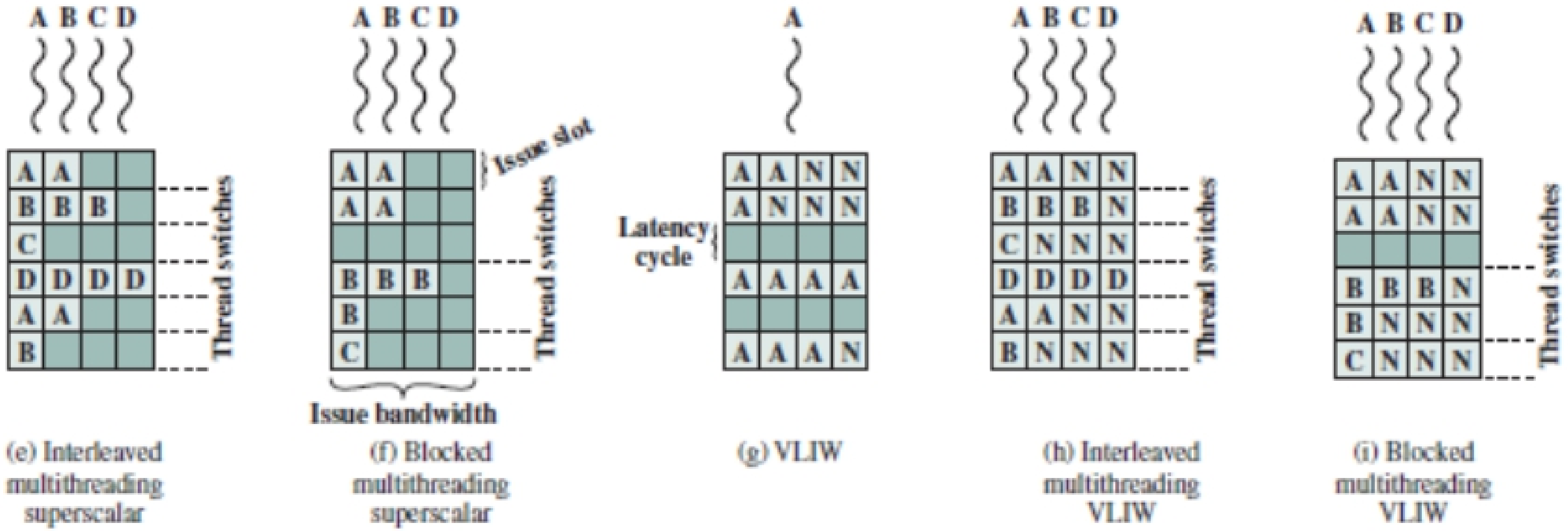
■ **Interleaved multithreading superscalar:** During each cycle, as many instructions as possible are issued from a single thread. With this technique, potential delays due to thread switches are eliminated. However, the number of instructions issued in any given cycle is still limited by dependencies that exist within any given thread.

■ **Blocked multithreaded superscalar:** Again, instructions from only one thread may be issued during any cycle, and blocked multithreading is used.

■ **Very long instruction word (VLIW):** A VLIW architecture, such as IA-64, places multiple instructions in a single word. Typically, a VLIW is constructed by the compiler, which places operations that may be executed in parallel in the same word. In a simple VLIW machine, if it is not possible to completely fill the word with instructions to be issued in parallel, no-ops are used.

■ **Interleaved multithreading VLIW:** This approach should provide similar efficiencies to those provided by interleaved multithreading on a superscalar architecture.

■ **Blocked multithreaded VLIW:** This approach should provide similar efficiencies to those provided by blocked multithreading on a superscalar architecture.



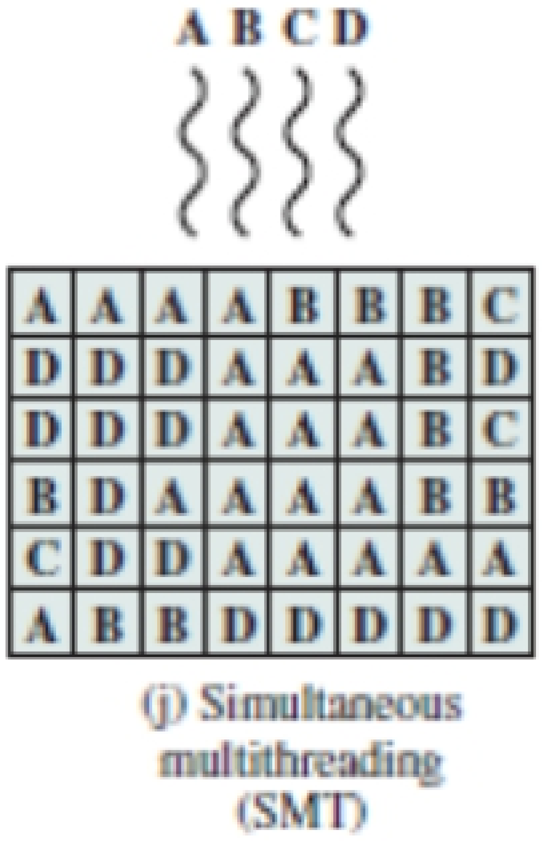
A B C D
Issue slot
Thread switches
Issue bandwidth
Latency cycle
(e) Interleaved multithreading superscalar
(f) Blocked multithreading superscalar
(g) VLIW
(h) Interleaved multithreading VLIW
(i) Blocked multithreading VLIW

Fig 7 (e-i): Multiple Thread Execution Approaches (contd.)

■ **Simultaneous multithreading:** If one thread has a high degree of instruction-level parallelism, it may on some cycles be able fill all of the horizontal slots. On other cycles, instructions from two or more threads may be issued. If sufficient threads are active, it should usually be possible to issue the maximum number of instructions on each cycle, providing a high level of efficiency.

■ **Chip multiprocessor (multicore):** Each core is assigned a thread, from which it can issue up to two instructions per cycle.

- A chip multiprocessor with the same instruction issue capability as an SMT cannot achieve the same degree of instruction-level parallelism. This is because the chip multiprocessor is not able to hide latencies by issuing instructions from other threads. On the other hand, the chip multiprocessor should outperform a superscalar processor with the same instruction issue capability, because the horizontal losses will be greater for the superscalar processor. In addition, it is possible to use multithreading within each of the cores on a chip multiprocessor, and this is done on some contemporary machines.



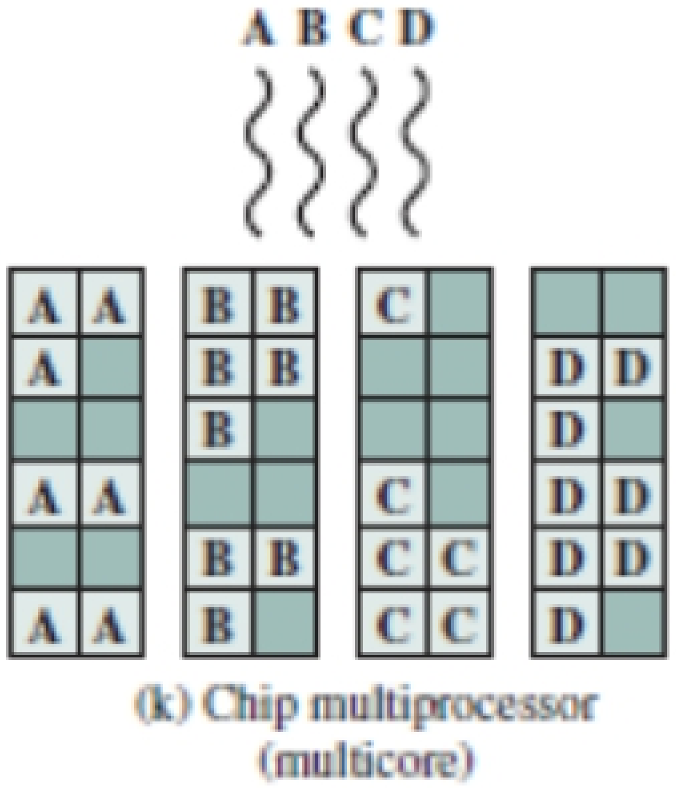


Fig 7 (j-k): Multiple Thread Execution Approaches (contd.)

LECTURE 35

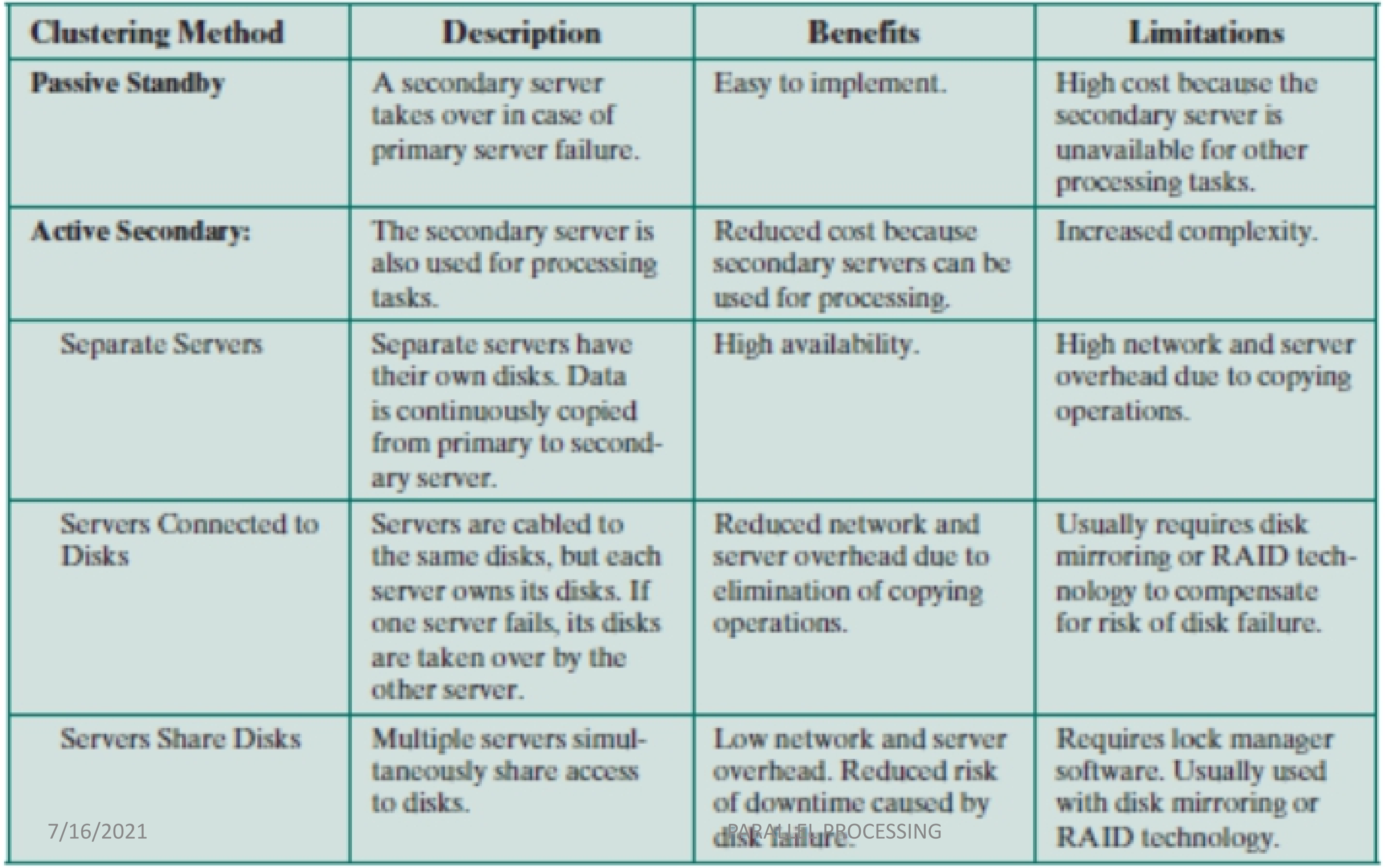
5. Clusters

- **Clustering** is an alternative to symmetric multiprocessing as an approach to providing high performance and high availability and is particularly attractive for server applications.
- A **cluster** as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine.
- Each computer in a cluster is typically referred to as a **node**.

Objectives or Design requirements

- ■ **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest standalone machines. A cluster can have tens, hundreds, or even thousands of machines, each of which is a multiprocessor.
- ■ **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- ■ **High availability:** Because each node in a cluster is a standalone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- ■ **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

Cluster Configurations



Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary:	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

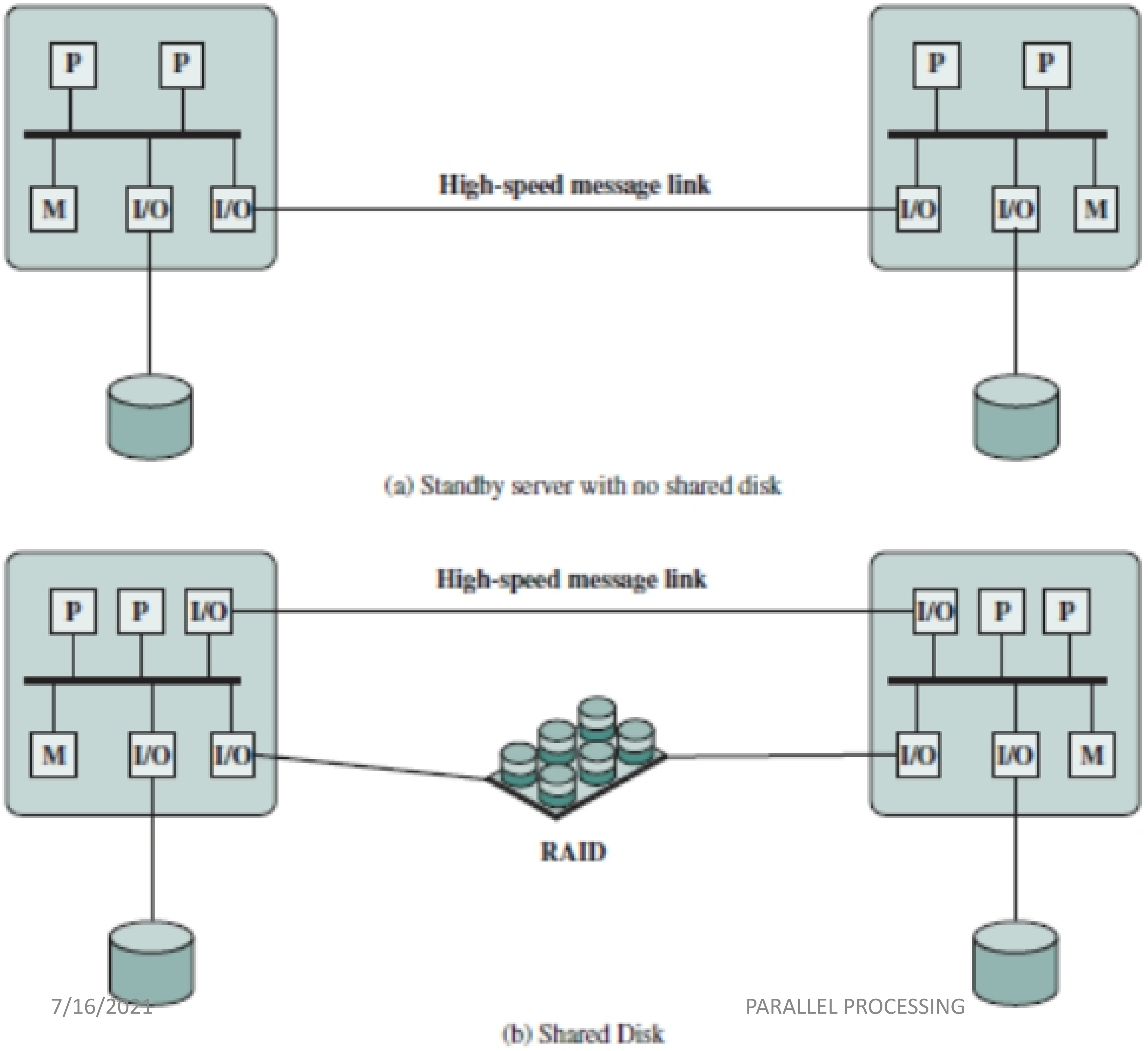


Fig 8: Cluster configurations

Operating System Design Issues

➢**Failure Management**: In case of failures, any one of the 2 approaches can be used: **highly available clusters** and **fault tolerant clusters**.

- A **highly available cluster** offers a high probability that all resources will be in service. If a failure occurs, such as a system goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.
- A **fault-tolerant cluster** ensures that all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

- The function of switching applications and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**.
- A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**.
- Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

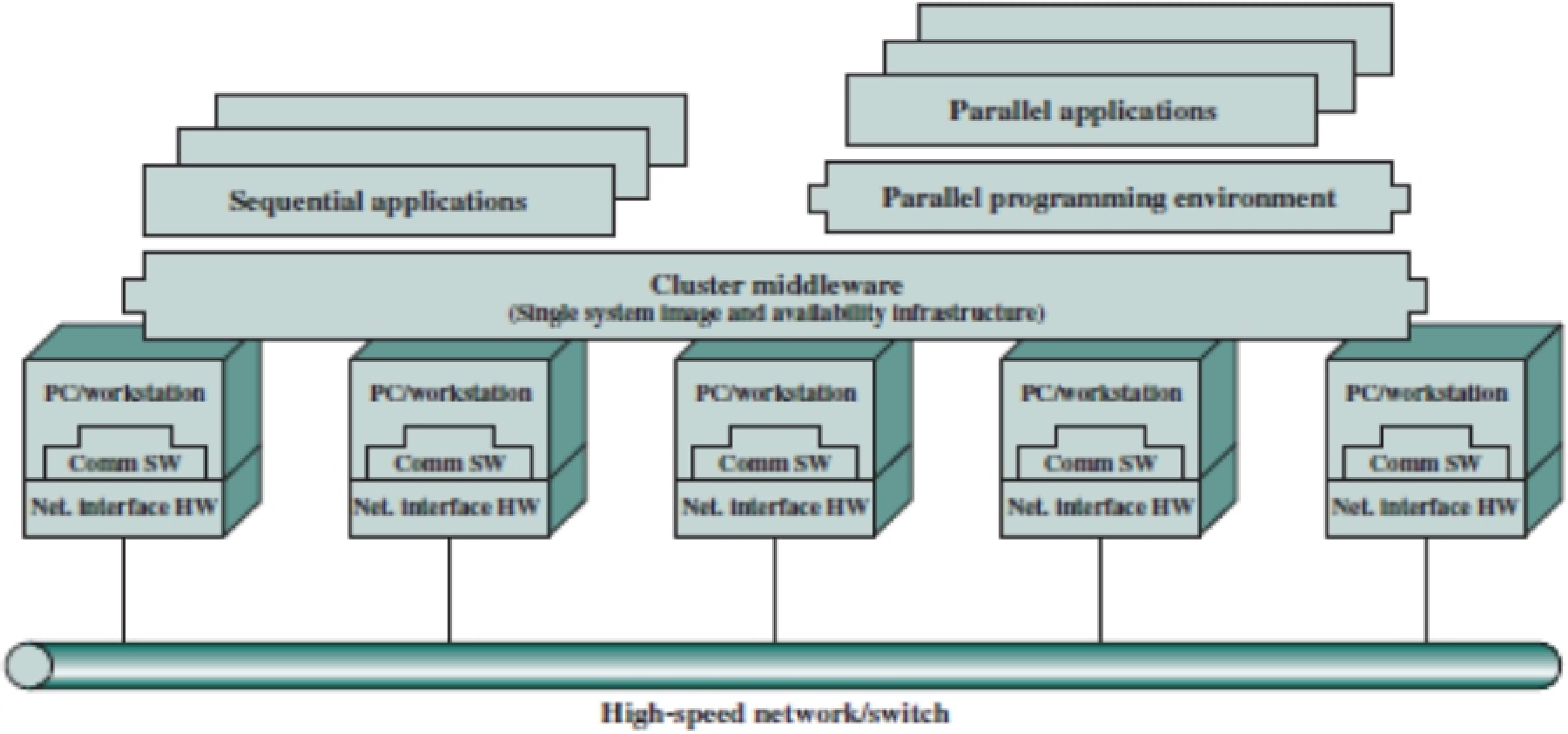
- **Load Balancing:** A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.
- **Parallelizing Computation**: By use of parallelizing compiler, parallelized application and parametric computing.

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed. In general, such compilers are difficult to develop.

- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster, and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios and then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an effective manner.

Cluster Computer Architecture

- The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently.
- In addition, a middleware layer of software is installed in each computer to enable cluster operation.
- The cluster middleware provides a unified system image to the user, known as a single-system image.
- The middleware is also responsible for providing high availability, by means of load balancing and responding to failures in individual components.



Parallel applications
Sequential applications
Parallel programming environment
Cluster middleware
(Single system image and availability infrastructure)
PC/workstation
Comm SW
Net. interface HW
High-speed network/switch

Fig 9: Cluster Computer Architecture

Desirable cluster middleware services and functions

- ✓ **Single entry point:** A user logs onto the cluster rather than to an individual computer.
- ✓ **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- ✓ **Single control point:** There is a default workstation used for cluster management and control.
- ✓ **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- ✓ **Single memory space:** Distributed shared memory enables programs to share variables.
- ✓ **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.
- ✓ **Single user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- ✓ **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- ✓ **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- ✓ **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- ✓ **Process migration:** This function enables load balancing.

Blade Servers

- A common implementation of the cluster approach is the blade server.
- A blade server is a server architecture that houses multiple server modules (“blades”) in a single chassis.
- It is widely used in data centers to save space and improve system management.
- Either self-standing or rack mounted, the chassis provides the power supply, and each blade has its own processor, memory, and hard disk.

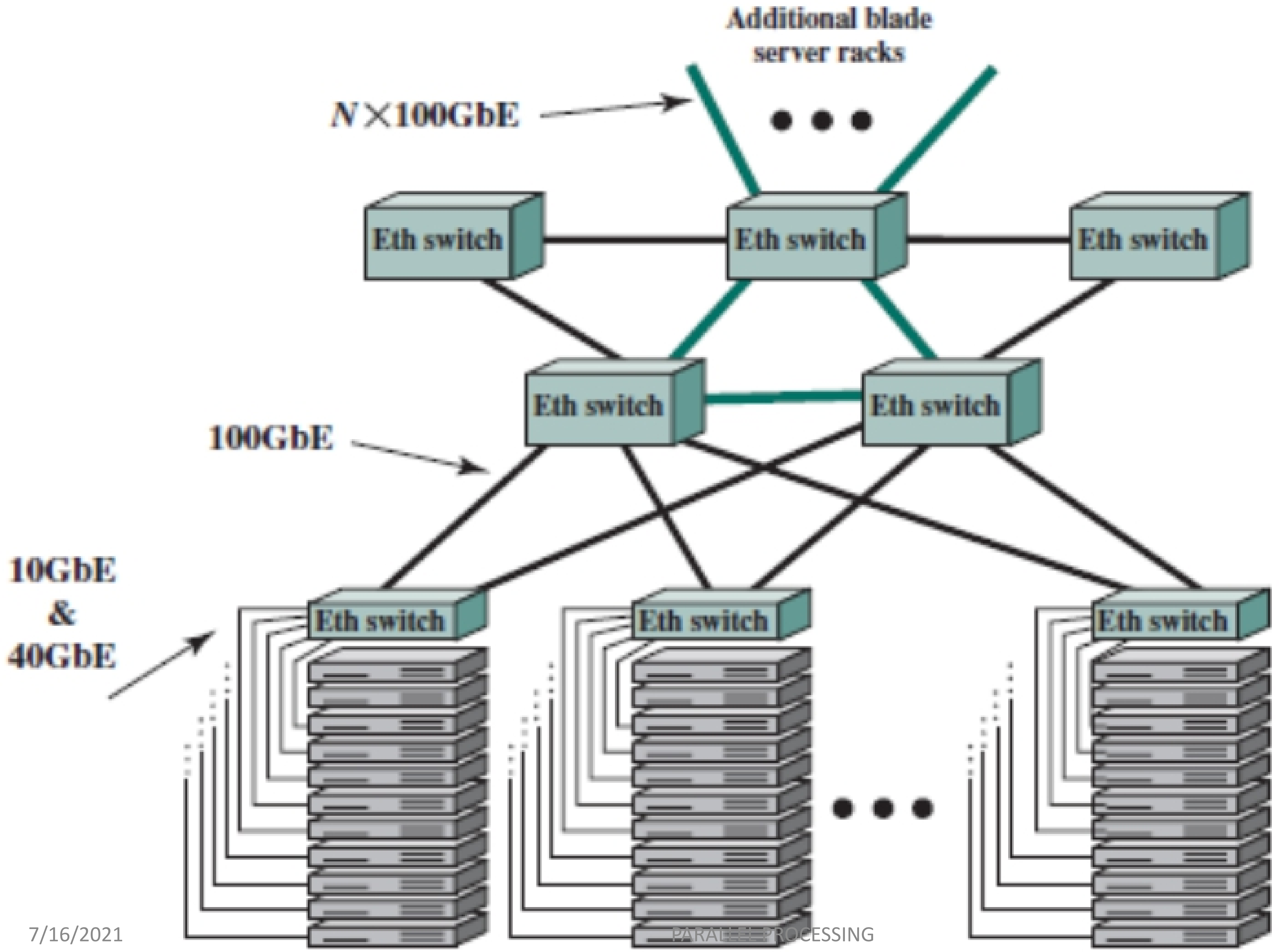


Fig 10: Example 100-Gbps Ethernet Configuration for Massive Blade Server site

Clusters vs SMP

- An SMP is easier to manage and configure than a cluster.
- The SMP is much closer to the original single-processor model for which nearly all applications are written.
- The principal change required in going from a uniprocessor to an SMP is to the scheduler function.
- Another benefit of the SMP is that it usually takes up less physical space and draws less power than a comparable cluster.
- A final important benefit is that the SMP products are well established and stable.
- The advantages of the cluster approach are likely to result in clusters dominating the high-performance server market.
- Clusters are far superior to SMPs in terms of incremental and absolute scalability.
- Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

6. Non-Uniform Memory Access

- **Uniform memory access (UMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same.
- **Nonuniform memory access (NUMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.
- **Cache-coherent NUMA (CC-NUMA):** A NUMA system in which cache coherence is maintained among the caches of the various processors.

- A NUMA system without cache coherence is more or less equivalent to a cluster.

- With an SMP system, there is a practical limit to the number of processors that can be used.
- An effective cache scheme reduces the bus traffic between any one processor and main memory.
- As the number of processors increases, this bus traffic also increases.
- Also, the bus is used to exchange cache-coherence signals, further adding to the burden.
- At some point, the bus becomes a performance bottleneck.
- Performance degradation seems to limit the number of processors in an SMP configuration to somewhere between 16 and 64 processors.
- The processor limit in an SMP is one of the driving motivations behind the development of cluster systems.
- However, with a cluster, each node has its own private main memory; applications do not see a large global memory. In effect, coherency is maintained in software rather than hardware.
- This memory granularity affects performance and, to achieve maximum performance, software must be tailored to this environment.
- One approach to achieving large-scale multiprocessing while retaining the flavor of SMP is NUMA.
- The objective with NUMA is to maintain a transparent system wide memory while permitting multiple multiprocessor nodes, each with its own bus or other internal interconnect system.

Organization

- There are multiple independent nodes, each of which is, in effect, an SMP organization. Thus, each node contains multiple processors, each with its own L1 and L2 caches, plus main memory.
- The node is the basic building block of the overall CC-NUMA organization. The nodes are interconnected by means of some communications facility, which could be a switching mechanism, a ring, or some other networking facility.
- Each node in the CC-NUMA system includes some main memory.
- From the point of view of the processors, however, there is only a single addressable memory, with each location having a unique system wide address.

- When a processor initiates a memory access, if the requested memory location is not in that processor’s cache, then the L2 cache initiates a fetch operation.
- If the desired line is in the local portion of the main memory, the line is fetched across the local bus.
- If the desired line is in a remote portion of the main memory, then an automatic request is sent out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus.
- All of this activity is automatic and transparent to the processor and its cache.
- In this configuration, cache coherence is a central concern.
- Each node must maintain some sort of directory that gives it an indication of the location of various portions of memory and also cache status information.

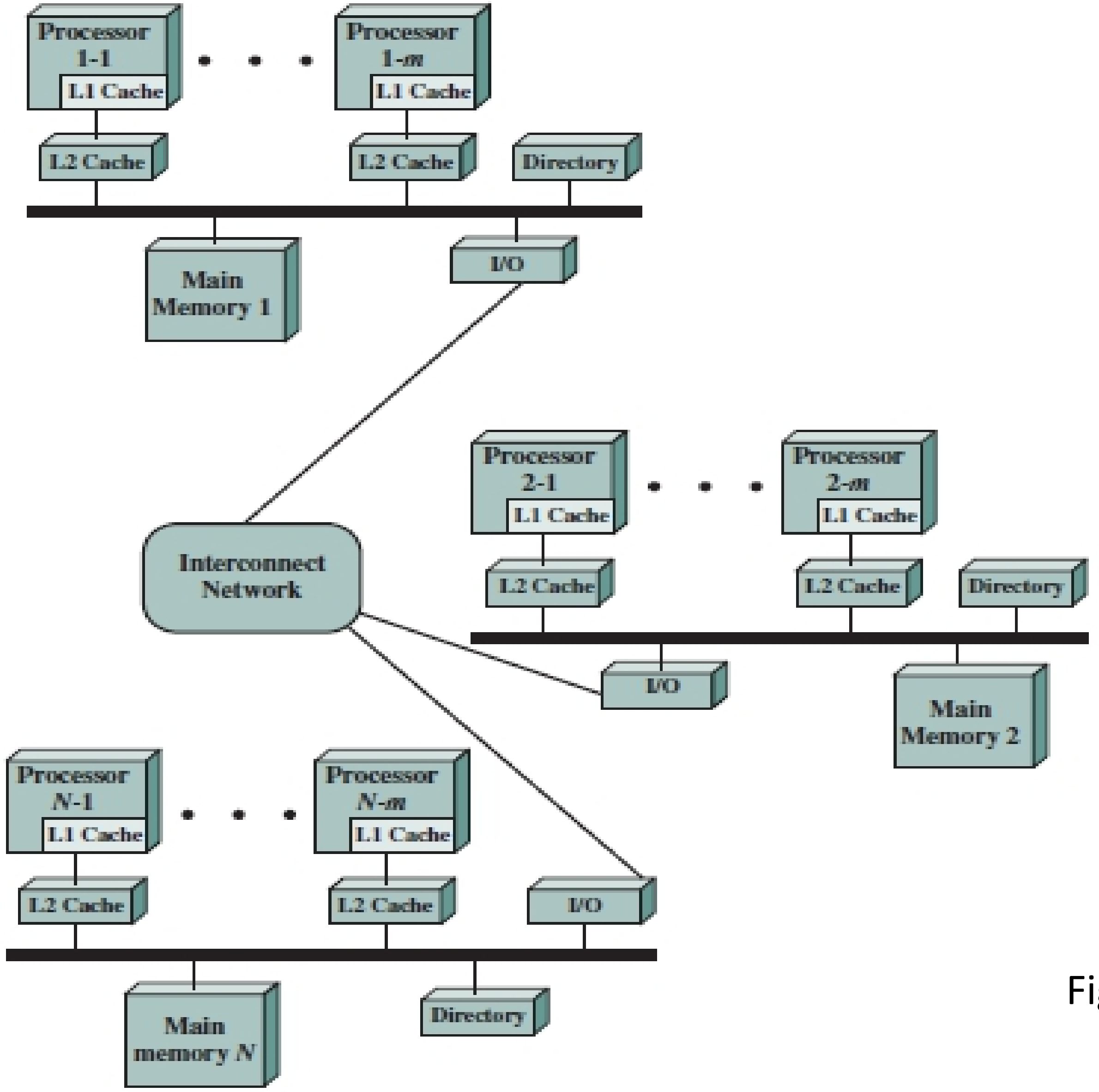


Fig 11: CC-NUMA Organization

Working Scheme

Suppose that processor 3 on node 2 (P2-3) requests a memory location 798, which is in the memory of node 1. The following sequence occurs:

1. P2-3 issues a read request on the snoopy bus of node 2 for location 798.

2. The directory on node 2 sees the request and recognizes that the location is in node 1.

3. Node 2's directory sends a request to node 1, which is picked up by node 1's directory.

4. Node 1's directory, acting as a surrogate of P2-3, requests the contents of 798, as if it were a processor.

5. Node 1's main memory responds by putting the requested data on the bus.

6. Node 1's directory picks up the data from the bus.

7. The value is transferred back to node 2's directory.

8. Node 2's directory places the data back on node 2's bus, acting as a surrogate for the memory that originally held it.

9. The value is picked up and placed in P2-3's cache and delivered to P2-3.

- As part of the preceding sequence, node 1’s directory keeps a record that some remote cache has a copy of the line containing location 798.
- Then, there needs to be a cooperative protocol to take care of modifications.
- For example, if a modification is done in a cache, this fact can be broadcast to other nodes. Each node’s directory that receives such a broadcast can then determine if any local cache has that line and, if so, cause it to be purged. If the actual memory location is at the node receiving the broadcast notification, then that node’s directory needs to maintain an entry indicating that that line of memory is invalid and remains so until a write back occurs. If another processor (local or remote) requests the invalid line, then the local directory must force a write back to update memory before providing the data.

Pros and Cons

- CC-NUMA can deliver effective performance at higher levels of parallelism than SMP, without requiring major software changes.
- With multiple NUMA nodes, the bus traffic on any individual node is limited to a demand that the bus can handle.
- However, if many of the memory accesses are to remote nodes, performance begins to break down.
- Disadvantages:
- First, a CC-NUMA does not transparently look like an SMP; software changes will be required to move an operating system and applications from an SMP to a CC-NUMA system. These include page allocation, process allocation, and load balancing by the operating system.
- A second concern is that of availability. This is a rather complex issue and depends on the exact implementation of the CC-NUMA system;

Avoiding performance breakdown in CC-NUMA system

- First, the use of L1 and L2 caches is designed to minimize all memory accesses, including remote ones. If much of the software has good temporal locality, then remote memory accesses should not be excessive.
- Second, if the software has good spatial locality, and if virtual memory is in use, then the data needed for an application will reside on a limited number of frequently used pages that can be initially loaded into the memory local to the running application.
- Finally, the virtual memory scheme can be enhanced by including in the operating system a page migration mechanism that will move a virtual memory page to a node that is frequently using It.

Thank You