Pipelining

Allows processing of several instructions
- It's mostly easily done for a sequence of instructions of the same/similar types that employ a single execution unit, eg floating point processor. However, all the same steps involved in the instruction processing by the CPU can be pipelined.

→ A pipeline system is compared to an assembly line on which many products are in various stages of manufacturing at the same time.

- In a non-pipelined CPU, instructions are executed in strict sequence as shown in Fig 1.1 a
- Pipelining permits the situation (shown in Fig 1.1 b) where each major step of instruction processing is assigned to and handled independently by a separate subunit (stage) of the CPU pipeline

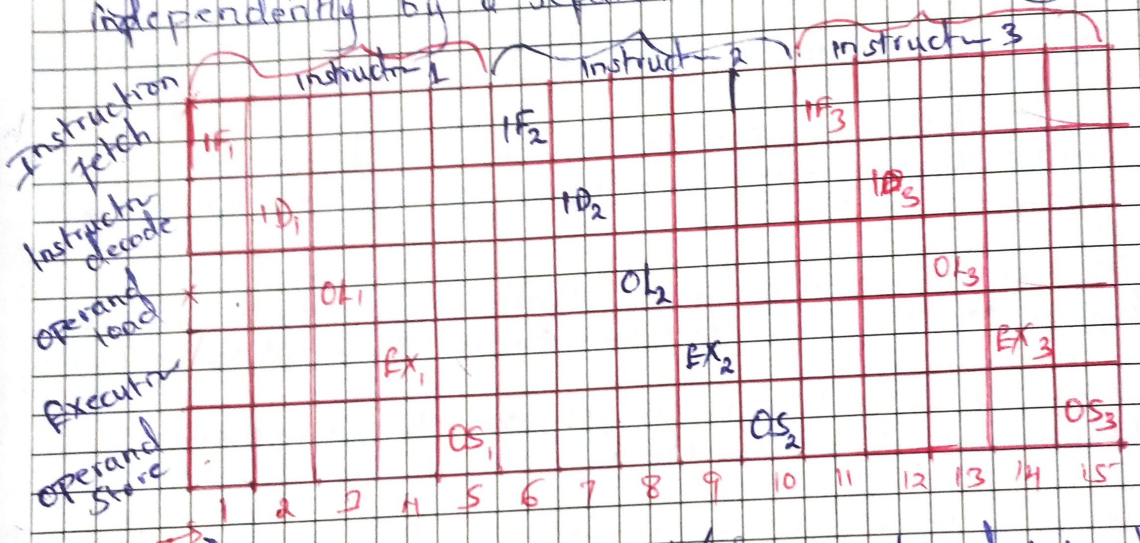


Fig 1.1 (a) Sequential/Non-pipelined instruction processing

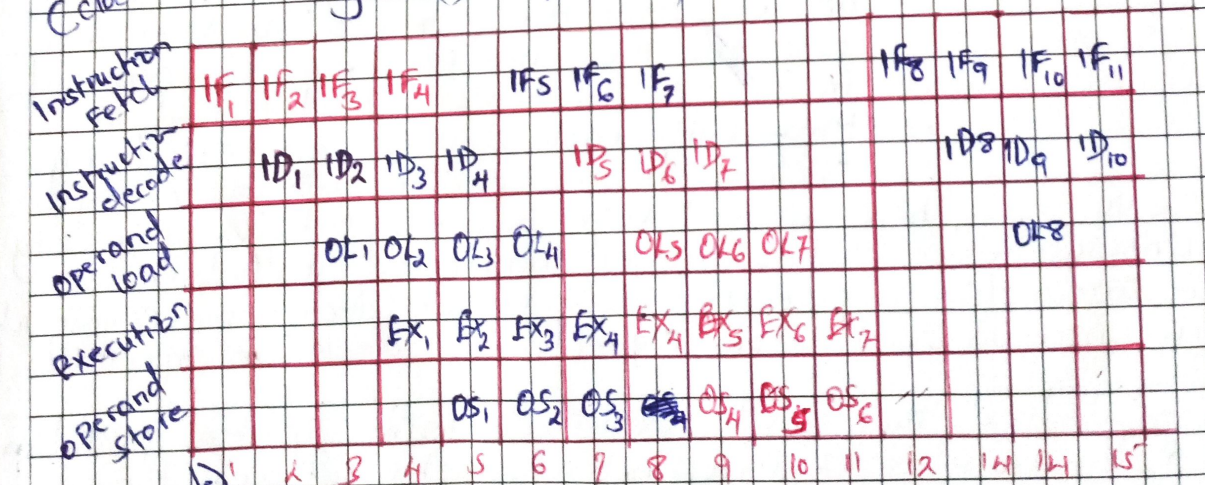


Fig 1.1 (b) Pipelined instruction processing

Pipelining reduces the number of clock cycles by more than half

(increases processing speed)

Power Conservation

- In this example, upto 5 instructions can be overlaped provided that the neceessary pipelined stages are available
- The performance-reducing delays occur as in the case of instruction I4 which must use the EX stage for 2 consecutive cycles
- A similar case occurs in the case of branch instructions like I7 where the outcome of the I7's Ex step must be known before the location of the next instruction I8 to be processed can be identified.

Super-Scaler

- A micro-processor's effective MIPS (million instructions per sec) rates can also be increased by replicating various instruction-processing circuits so that several instructions can be in the same processing phase at the same time.
- This makes it possible to start processing or issue 2 or more instructions simultaneously or in parallel i.e the instructions can be totally overlaped
- CPUs with this capability are said to be Super-Scaler
- Pipelining & super-scaler design are both instances of instruction level parallelism
- the logic circuits needed to deal with parallelism of this kind are considered complexity to the CPUs program control & execution in MIPS
- 2 fundamental ways in which components can be composed to create parallel computing structures are:

① Temporal Parallelism :- Involves partitioning the processing task into a number of steps which when applied sequentially to each unit of info produces the same results as the original task. i.e the task is partitioned in time with each step of the task being applied to a separate unit of the info

Example
- Assembly line in manufacturing
- Pipeline structuring

② Spatial Parallelism :- The component used to carry out the processing task isn't subdivided but is instead replicated so that each unit of information is processed by its own dedicated component
- The task space is parallelized e.g checkout desks in supermkts

The amount of parallelism that can be exploited using spatial parallelism depends only on the number of independent tasks whereas the amount of parallelism that can be exploited using temporal parallelism depends on the divisibility of the task being parallelized.

SMPs
Symmetric
cluster
Non-uniform Memory Access

2.0 Parallel Processing

Intro

- Traditionally, the computers have been built as a sequential machine
- Most comp programming languages requires the programmer to specify algorithm as sequences of instructions.
- Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instructions, fetch operands, store results, perform operations)

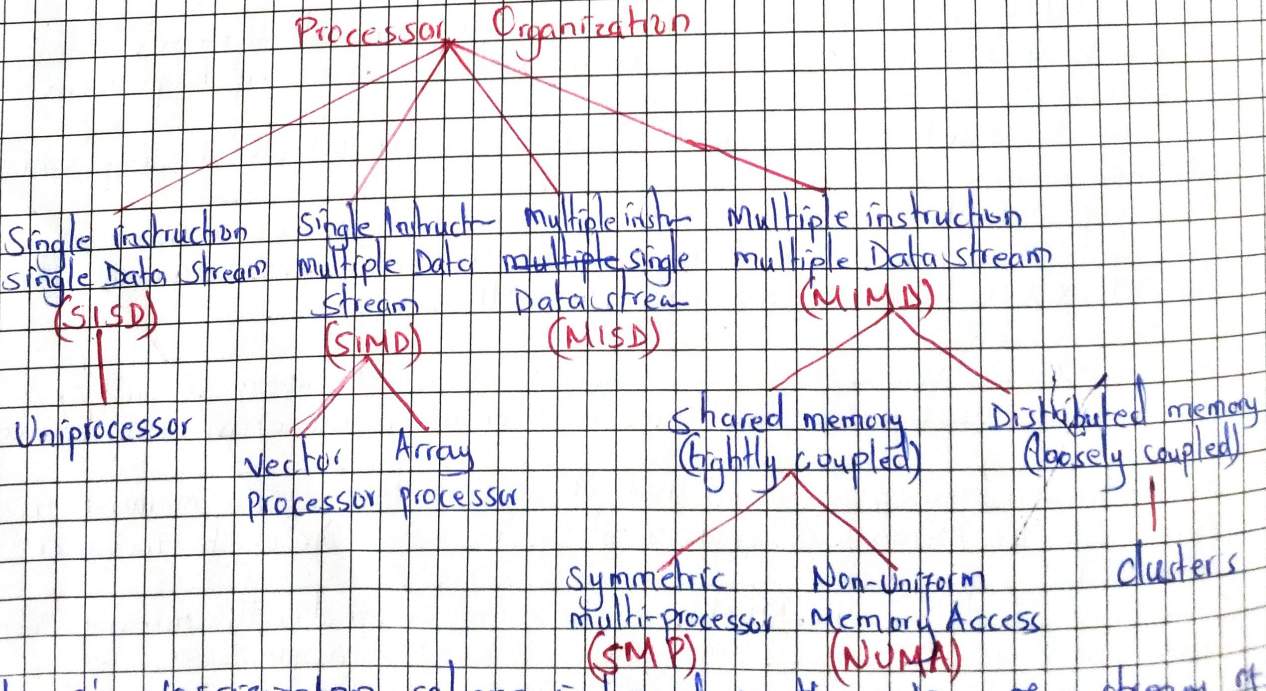
NB This view of the computer has never been entirely true.

- At the micro-operations level, multiple control signals are generated at the same time
- Instruction pipelining atleast to the extent of overlaping fetch and execute operations has been around for a long time. Both of these are examples of performing functions in parallel
- This approach is taken further with super-scaler organization which exploits instructions level parallelism
- As Comp tech has evolved, & as the cost of comp hw developed, computer designers have more and more opportunity of parallelism usually enhanced performance and in some cases increase affordability.

Multiple Processor Organizations

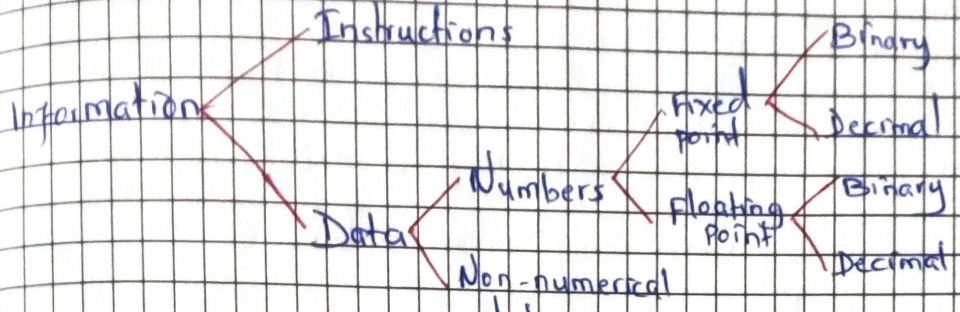
Types of parallel processors systems:-

- Flynn's Taxonomy :- It consists of 4 categories of computer systems as depicted in fig 2.1



- Flynn's classification scheme is based on the notion of a stream of information. 2

Types of information given into the processor (Instruction & Data)



① SISD stream :- a single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category

② SIMD stream :- A single machine instruction controls the simultaneous execution of a number of processing units in a lock step basis. Each processing element has an associated memory so that each instruction is executed on a different set of data by the different processors
- Vector and array processors fall into this category.

③ MISD stream :- a sequence of data is transmited to a set of processors, each of which executes a different sequence of instruction
- The structure is not commercially implemented.

④ MIMD stream :- a set of processors simultaneously executes diff instruction sequences on different data sets.
SMPS, clusters and NUMA falls in this category.

- With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform appropriate data transformation.
- MIMDs can be further subdivided by the means in which the processors communicate.
- If the processors share a common memory the each processor access a program and data stored in each memory and the processors communicat with each other via that memory eg SMP

- In an SMP multiple processors share a single memory or pull of memory by means of a shared bus or other interconnection mechanism

- The distinguishing feature is that the access time of the memory in any region of the memory is approximately the same for each processor

- A more recent dvt is the NUMA, as the name suggests, the memory access time for different regions may differ for a NUMA processor

- The collection of independent processors or SMPS may be interconnected to form a cluster.

2.1.2 Parallel Organization.

2.2 SMP:

Until fairly recently, virtually all single-user personal comps and most workstations contained a single general-purpose microprocessor.

As the demand for performance increased & the cost of microprocessors continues to drop, vendors have introduced systems with an SMP organization.

SMP refers to a comp hardware architecture and also to the OS behavior that reflects that architecture.

- SMP can be defined as a stand alone comp with the following attributes
  - There 2/more similar processors of comparable capabilities
  - These processors share the same main memory & I/O facilities & are interconnected by a bus or any other internal connection scheme such that mem access time is approximately the same for each processor.
  - All processors share access to I/O devices, either through the same channels or diff channels that provide paths to the same device.
  - All processors can perform the same function (hence the term symmetric)
  - The system is controlled by an O/S that provides an interaction btn processors & their programs at the job, task, files & data element levels.

- The O/S of an SMP schedules processes/threads across all of the processors
  - Process:- is an instance of a program running in a comp
  - Thread:- is a dispatchable unit of work within a process

- An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:-

(1) Performance:- if the work to be done by the comp can be organized so that some portions of the work can be done in parallelism, then a system with multiple-processors will yield greater performance than one with a single-processor at the same time. Fig 2.3

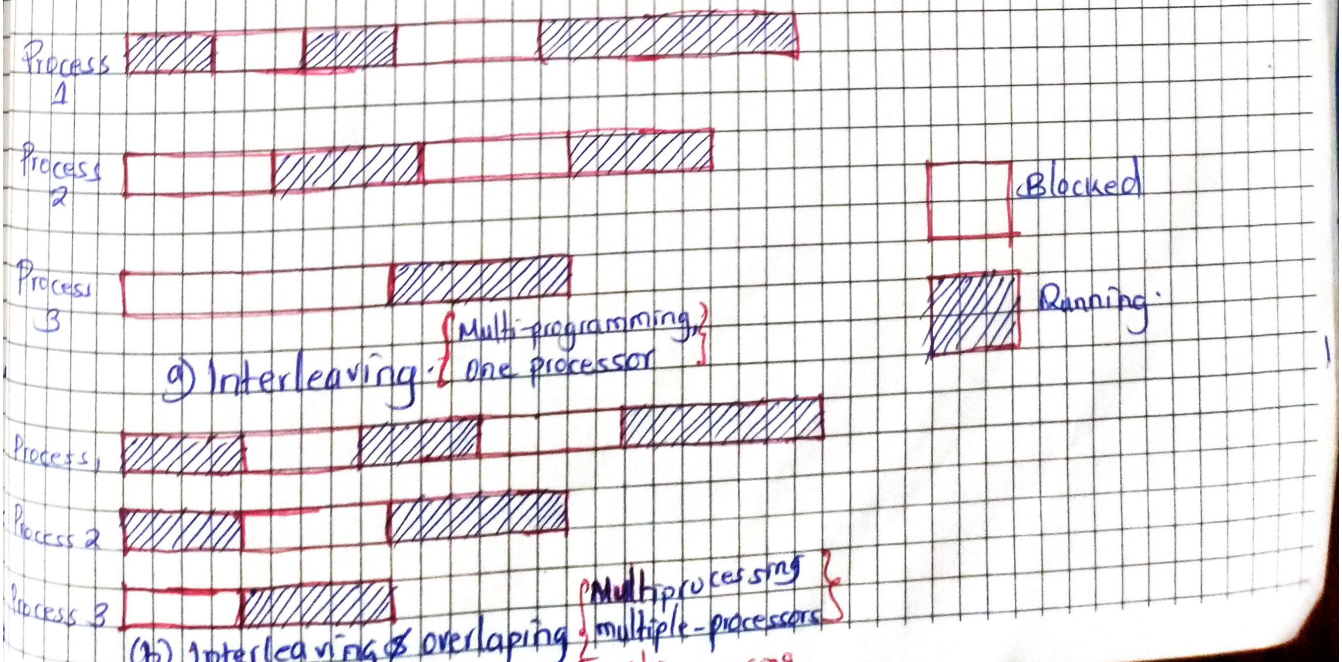


Fig 2.3 multiprogramming & multiprocessing.

(ii) Availability:- In an SMP since all processors can perform the same functions, the failure of a single processor doesn't halt the machine instead the system can continue to function at reduced performance.

(iii) Incremental Growth:- The user can enhance performance of a system by adding an additional processor.

(iv) Scaling:- Vendors can offer a range of products with diff price and performance designed
xtics based on the number of processors configured in the system.

⇒ There are potential greater than guaranteed benefits.
- The OS must provide tools and functions to exploit the parallelism in an SMP system.
- An attractive feature of an SMP is that the existence of multi-processor is transparent to the user.
- The OS takes care of processing & scheduling of threads/processes of individual processors

Organization of the SMP system

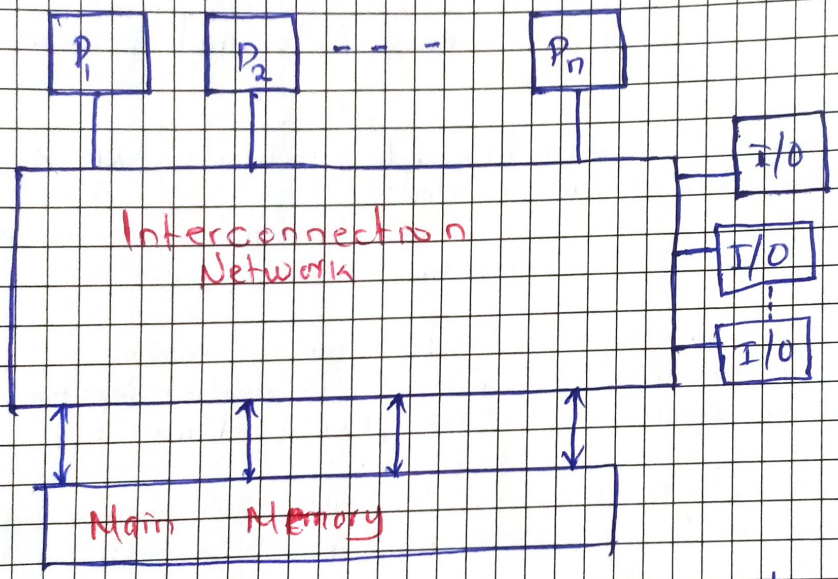
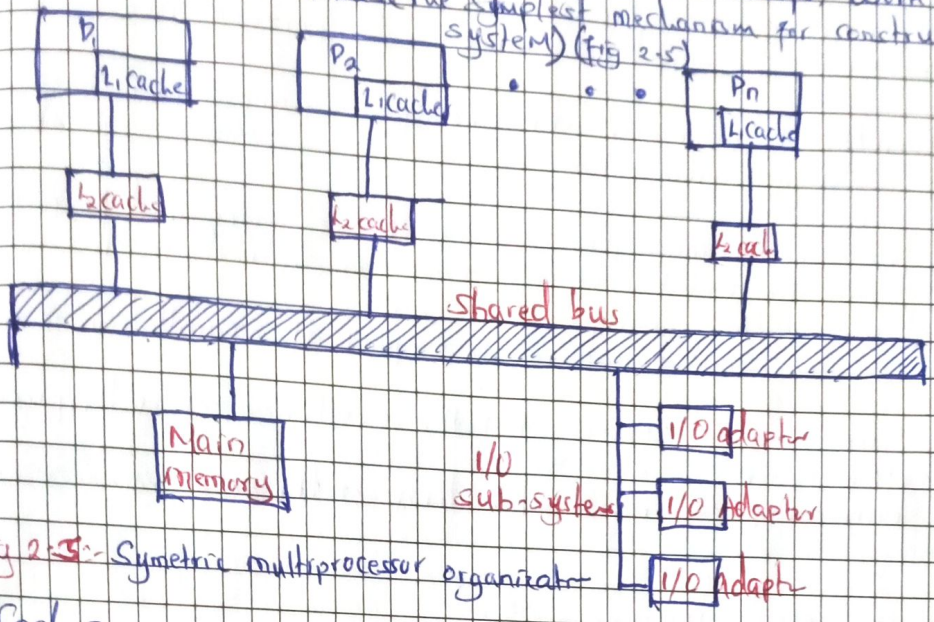


Fig 2.4 Generic block diagram of tightly coupled multi-processor

- Fig 2.4 depicts in general terms, the organization of a multi-processor system
- There are 2/more processors each processor is self-contained including ALU, control unit, registers & typically one or more levels of cache.
- Each processor has access to a shared main memory & the I/O devices through some form of interconnection mechanism.
- The processors can communicate to each other through memory (message & status info. left in compound data areas)
- It is often organized may also be possible for processors to exchange signals directly.
- The mem is often organized so 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

Time-sharing - deadlocks

The most common organization for personal comps, work stations & servers is the time-shared bus (the simplest mechanism for constructing a multiprocessor system) (fig 2.5)



Advantages
- simplicity
- flexibility
- Reliability

disadvantages
- Bottlenecks
- Poor performance

Fig 2.5:- Symetric multiprocessor organization

~~Each processor~~
- The structure & interfaces are basically the same as for a single-processor system that uses a bus-interconnection.
- The bus consists of control, address & data lines. To facilitate DMA transfers from I/O devices, the following features are provided:

i) Addressing :- It must be possible to distinguish modules on the bus to determine the source & destination of data.

ii) Arbitration :- Any I/O module can temporarily function as master. A mechanism is provided to arbitrate ~~request~~ competing request from bus controls using some priority scheme

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

- The uni-processor features are directly usable in an SMP connection
- In SMP connection there multiple processors as well as multiple I/O systems

Attractive features of Bus organization (Advantages)

- Simplicity :- simplest approach to multi-processor organization, the physical interface addressing & time sharing/arbitration of each processor remain the same as in the single processor system.
- Flexibility :- it is generally easy to expand the system by attaching more processor to the bus.
- Reliability :- the bus is essentially a passive medium & the failure of any attached device should not cause failure of the whole system.

Drawbacks:
- Performance :- all mem references pass through the common bus thus the bus cycle time limits performance of the system.

To improve performance, it is desirable to equip each processor with a cache mem, this would reduce the number of bus access dramatically.

$2\pi \equiv 360°$ ∴

$$T = \frac{N \times CPI}{f(\times 10^6)}$$ f is in megahertz MHz

$$\left(10^6 \times \frac{f}{CPI}\right) = \frac{N}{T} = \text{rate of execution} = (\text{mips})$$

- Typically, work station, PC, MPCs has 2 levels of cache with level 1 cache internal & L2 cache either internal or external.
- Some processors has L3 cache
- The use of caches introduced some new design considerations
- Since each local cache contains an image of a portion of memory, if a word is altered in one cache it could invalidate a word in another cache
- To prevent this, the other processors must by alerted that an update has taken place
- a problem called cache coherence problem is addressed typically in H/W rather than in the OS system.

Multi-processor Operating System Design Considerations:

- An SMP OS manages processor & other comp resources so that the user perceives a single OS controlling system resources.
- In fact such a configuration should appear as a single-processor multi-programming system.
- In both SMP & uniprocessor cases, multiple processes may be active at one time, and it is the responsibility of the OS to schedule their execution and to allocate resources.
- A user may construct apps that use multiple processor/multiple threads within processors without regard to whether a single-multiple processors will be available, thus a multi-processor OS must provide all the functionality of a multi-programming system plus additional features to accomodate multiple processors
- Among the design key issues are:

① + **Simultaneous concurrent processes:-** OS routines need to be re-entrant to allow several processors to execute the same instruct stream (IS) code simultaneously.
- with multiple processors executing the same or diff parts of the OS, OS tables & mngt structures must be managed properly to avoid deadlocks or invalid functions/operations.

② + **Scheduling:-** any processor may perform scheduling & acts as a master. 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 IO resources, care must be taken to provide effective synchronization.
This is a facility that enforces mutual exclusion & event ordering.

④ **Memory Management.**
- Mem mngt on a multi-processor must deal with all the issues found on uniprocessor machines
- The OS needs to exploit the available H/W parallelism e.g multi-ported memories to achieve the best performance.

⑤ + **The paging mechanisms:** on diff processors must be coordinated to enforce consistency when several processors share a page/segment & to decide on page replacement.

reduced

6. **Reliability & fault tolerance:-** The OS should provide graceful degradation in the phase of processor failure. The scheduler & other portions of the OS must recognize the loss of the processor & restructure the mngt tables accordingly.

Mainframe SMP

Most PC & work station SMPs use a bus interconnection strategy as shown in fig 2.5.

It is instructive to look at an alternative approach which is used for a recent implementation of IBM z series Mainframe family.

This family of systems spans range from a uniprocessor with one main mem card to the high-end system with 48 processors & 8 memory cards.

The key components of the configuration are shown in fig 2.6.

① **Dual core processor chip:-** each processor chip includes 2 identical ~~server~~ central processors (CPs). The CP is a CISC (Complex Instruction set Comps). Each CP includes a 256 kb L_1 instruction cache and a 256 kb L_1 data cache.

② **L_2 cache:-** each contains 32 MB. The L_2 caches are arranged in clusters of 5 with each supporting 8 processor chips & providing access to the entire memory space.

③ **System control element (SCE):-** arbitrates system communication & has the central role in maintaining cache coherrence.

④ **Main Memory Store Control (MSC):-** interconnect the L_2 caches & the main memory.

⑤ **Memory card:-** each card holds 32 GB of memory, maximum configuration of 8 cards for a total of 256 GB.
Memory cards interconnects the SMs via synchronous memory interfaces (SMIs)

⑥ **Memory Bus Adapter (MBA):-** provides interface to various types of I/O channels. traffic to-and-from the channels go directly to the L_2 cache

- The microprocessor in the z990 is relatively ~~&~~ common compared with other modern processors
- It executes instructions in strict architectural order. However, it makes up for this by having pipeline (shorter) & much larger caches
- The L_2 & SCE connects with corresponding elements in other books in a ring configuration.

Interesting features in z990 SMP config:-

① Switched interconnection
② Shared L_2 caches
③

① **Switched interconnection:-** a single shared bus is a common arrangement for SMPs for PCs & works stations (Fig 2.5)
With this arrangement, the single bus become a bottleneck affecting the scalability (Ability to scale larger sizes) of the design.
The z990 deals with this problem in 2 ways

reading: L2 Assignment :- Cache Coherence
- MESI Protocol

- (i) Main memory is split into 4 cards each with its storage controller that can handle memory accesses at high speed.
The average traffic load is cut by 4 of the independent paths to and from parts of the memory. Each book include 2 memory cards for a total of 8 cards across a max configuration.

(ii) The connection from processors (L2 caches) to a single mem card is not in the form of a shared bus but rather point-to-point links. Each processor chip has a link to each of the L2 caches on the same book and each L2 cache has a link via the MSC to each of the 2 mem cards on the same book.

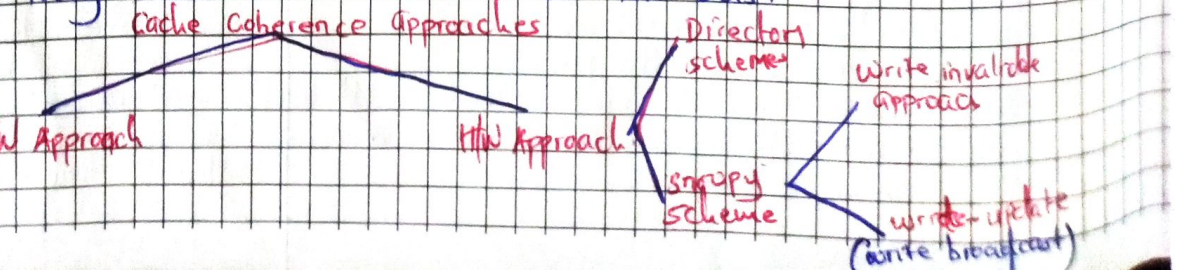
- Each L2 cache only connects to the 2 mem cards on the same book.
- The system controller provides links to the other books in the configuration, so that all of main memory is accessible by all the processors.
- Point-to-point links, rather than a bus also provides connections to I/O channels. each L2 cache on a book connect to each of the MBAs for that book. The MBAs in turn connects to the I/O channels.

Shared L2 Caches.

- In a typical 2-level cache scheme for SMP, each processor has a dedicated L1 & L2 caches.
- In recent years, interest in the concept of a shared L2 cache has been growing. In an earlier version of its mainframe SMP, known as G3, IBM made use of dedicated L2 caches. In its later versions (G4, G5, z900) a shared L2 cache is used. 2 considerations dictated this change:
  (i) - In moving from G3 to G4, IBM doubled the speed of the microprocessors. if the G3 organization were retained, the significant increase in bus traffic would have occurred. At the same time it was designed to reuse as many components as possible.
  - Without a significant bus update, the bus system network would become a bottleneck.
  (ii) - Analysis of a typical mainframe workload revealed a large degree of sharing instructions and data among processors

- These configurations led the G4 designing team to consider the use of 1 or more L2 caches each processor having a dedicated on-chip L1-cache

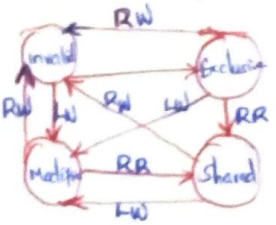
- At first glance, sharing an L2 cache might seem a bad idea
- Accessed memory from processors should be slower because the processors must now access/contend to a single L2 cache. However, if a sufficient amount of data is in fact shared by a multiple processor, then a shared cache can increase throughput greater than reterdity.
- Data that are shared and stored in cache are obtained more quickly that if they must be obtained over the bus.



RW :- Remote "
LW :- Local "

MIPS → Million Instructions Per Second.

data that's not present in the cache



Multithreading & Chip Multiprocessors:

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$\text{MIPS rate} = f \times IPC \quad \text{(Instruction Per Cycle)}$$

where: f = processor clock frequency (MHz)
IPC = average number of instructions executed per cycle

$$MIPS = IPS \times 10^6 = \frac{f}{CPI} (MHz)$$

Accordingly, designers have persued the goal of increased performance on two fronts:-

1- Increasing clock frequency
2- Increasing the number of instructions executed i.e number of complete instructions in a processor cycle.

- Designers have increased IPC by using an instruction pipeline & then by using multiple parallel instruction pipelines in the super-scalar.

- With pipelined & multiple-pipelined designs, the principal problem is to maximize the utilization of the pipeline stages.
- To improve throughput designers have created ever more complex mechanisms eg executing some instructions in a different order from the way they occur in the instruction scheme & beginning execution of instructions that may never be needed.

- An alternative approach called multithreading allows for a high degree of instruction level parallelism without increasing circuit complexity/power consumption.
- The instruction stream is divided into several smaller streams.

Implicit & Explicit multithreading:

Process:- an instance of a program running on a comp
It embodies 2 key x-tics.
(i) - Resource ownership:- a process includes a virtual address space to hold the process image i.e the collection of program data stack & attributes that define the process.
- From time-to-time the process may be allocated control & ownership of the process eg main mem I/O channels & devices.

Cache Coherence:

A situation where multiple processor cores share the same memory hierarchy, but have their own L1 data & instruct caches

Cache coherence problem
Cache from Core1 & Core2 has a copy of data of same value read from the memory but whenever a given core is performing instruction store while other subsequently instruction load, the value read becomes different & thus must be propagated to the copy of variable in the reading core.

Cache Coherence protocol
- set of rules that governs how multiple caches interact in order to solve cache coherence

Approaches:-
* Validation-based cache coherence protocol (commonly known as MESI)
- Solves the problem by ensuring that as soon as a core requests to write to a cache block, that core must invalidate (remove) the copy of the blocks in any other core cache that contains the blocks. The requesting core then has the only copy of the cache blocks & can make modifications to its contents. Later, when any other core attempts to read the blocks, it will experience a cache miss & must obtain the new data from the core that modified the data.

MESI → Named after the 4 states that a cache block in an L1 cache can have:
Modified:- dirty with respect to shared mem hierarchy. Make changes at will.
Exclusive:- clean:- if the owning core wants to write to the data, it can change data state to modified without consulting any other cores
Shared:- clean:- read-only: can read a block but only write after transition to exclusive state.

O - over
P - opportunities
R - Repeatedly

6.4

• Multithreading → SMT

Chip

✓ Vertical loss ✓

(1) 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) & a dispatching priority & is the entirety that is scheduled & dispatched by the OS.

(2) ⇒ Process switch :- an operation that switches the processor from 1 process to another by saving all other process control data, register & other ~~operations~~ info for the first & replacing them with the process info for the 2nd.

(3) ⇒ Thread :- is a dispatchable unit of work within a process. It includes a processor context (PC & stack pointer) and its own data area for a stack (to enable sub-routine branching)
- A thread executes sequentially & is interruptable so that the processor can turn to another thread.

(4) ⇒ Thread switch :- the act of switching processor control from one thread to another within the same process.
- Typically, this type of switch is less costly than a process switch.

⟹ Thus a thread is concerned with scheduling & execution whereas a process is concerned with both, scheduling & execution & resource ownership.

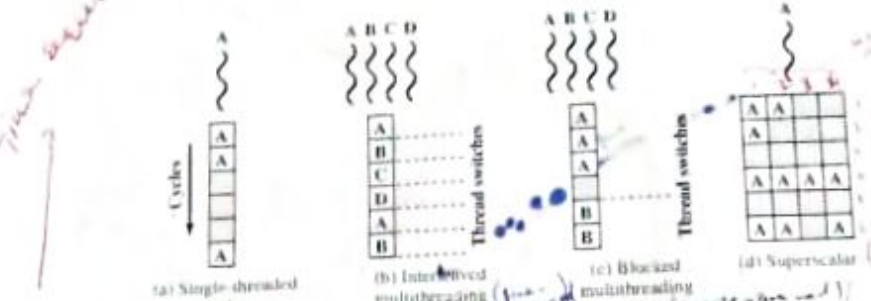
⟹ The multiple threads within a process share the same resources. Consequently, a thread switch is much less time consuming than a process switch.

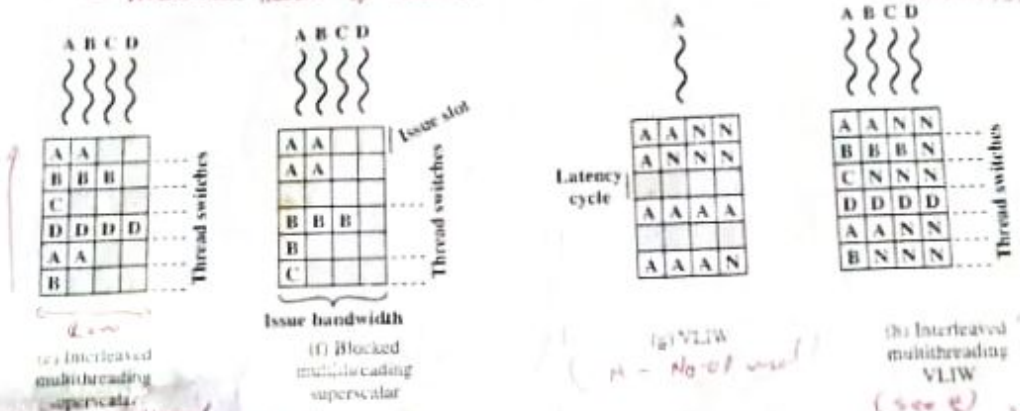
⇒ Distinction btn user-level threads & Kernel-level threads :-
* User-level threads are visible to the application program while kernel-level threads are only visible to the OS
- Both can be referred to as explicit threads defined in SW.

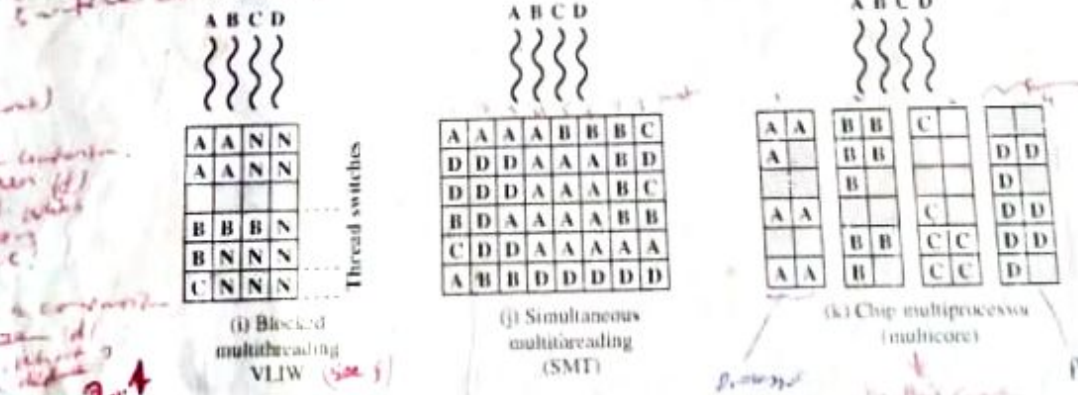
- Most experimental processors & all the commercial processors so far have used explicit multithreading. These systems concurrently execute instructions from diff explicit threads either by interliving instructions from diff threads on shared pipelines or by parallel execution on parallel pipelines

- Implicit multithreading refers to 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 ~~Stat~~ compiler

Approaches to Explicit Multithreading.







Approaches to Executing Multiple Threads

(c) **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.

Figure 17.5c shows a situation in which the time to perform a thread switch is one cycle, whereas Figure 17.5b shows that thread switching occurs in zero cycles. In

Word length

A word is a unit of information of some fixed length n represented in a binary system. Word size is typically a multiple of 8, common CPU word sizes being 8, 16, 32, and 64 bits

P.T.O

- **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.[2]
- **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.

Thus, 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. Traditional operating systems, such as earlier versions of UNIX, did not support threads. Most modern operating systems, such as Linux, other versions of UNIX, and Windows, do support thread. A distinction is made between user-level threads, which are visible to the application program, and kernel-level threads, which are visible only to the operating system. Both of these may be referred to as explicit threads, defined in software.

All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to 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. In the remainder of this section we consider explicit multithreading.

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. What is achieved is thread-level parallelism, which may provide for greatly improved performance when married to instruction-level parallelism.

Broadly speaking, there are four principal approaches to multithreading:

- **Interleaved multithreading:** This is also known as **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

[2] The term *context switch* is often found in OS literature and textbooks. Unfortunately, although most of the literature uses this term to mean what is here called a process switch, other sources use it to mean a thread switch. To avoid ambiguity, the term is not used in this book.

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, as illustrated in Figure 17.8. This is true if a fetched instruction triggers the thread switch and must be discarded from the pipeline [UNGE03].

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.

More opportunities for parallel execution are available if the processor can issue multiple instructions per cycle. Figures 17.8d through 17.8i illustrate a number of variations among processors that have hardware for issuing four instructions per cycle. In all these cases, only instructions from a single thread are issued in a single cycle. The following alternatives are illustrated:

- **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*.
- **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, as previously discussed. 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 (Figure 17.8g), 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.

The final two approaches illustrated in Figure 17.8 enable the parallel, simultaneous execution of multiple threads:

- **Simultaneous multithreading:** Figure 17.8i shows a system capable of issuing 8 instructions at a time. 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):** Figure 17.5c shows a chip containing four processors, each of which has a two-issue superscalar processor. Each processor is assigned a thread, from which it can issue up to two instructions per cycle. We discuss multicore computers in Chapter 18.

3.1 MULTITHREADING AND CHIP MULTIPROCESSORS

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$\text{MIPS rate} = f \times IPC$$

where f is the processor clock frequency, in MHz, and IPC (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle. As we have seen in earlier chapters, designers have increased IPC by using an instruction pipeline and then by using multiple parallel instruction pipelines in a superscalar architecture. With pipelined and multiple-pipeline designs, the principal problem is to maximize the utilization of each pipeline stage. To improve throughput, designers have created ever more complex mechanisms, such as executing some instructions in a different order from the way they occur in the instruction stream and beginning execution of instructions that may never be needed. But as was discussed in Section 2.2, this approach may be reaching a limit due to complexity and power consumption concerns.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called multithreading. In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel.

The variety of specific multithreading designs, realized in both commercial systems and experimental systems, is vast. In this section, we give a brief survey of the major concepts.

Implicit and Explicit Multithreading

The concept of thread used in discussing multithreaded processors may or may not be the same as the concept of software threads in a multiprogrammed operating system. It will be useful to define terms briefly:

- **Process:** An instance of a program running on a computer. A process embodies two 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.

of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.

- **Blocked multithreading:** This is also known as **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:** In this case, the entire processor is replicated on a single chip and each processor handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design. This is referred to as multicore; we examine this topic separately in Chapter 18.

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.

Figure 17.8, based on one in [UNGE02], illustrates some of the possible pipeline architectures that involve multithreading and contrasts these with approaches that do not use multithreading. Each horizontal row represents the potential issue slot or slots for a single execution cycle; that is, the width of each row corresponds to the maximum number of instructions that can be issued in a single clock cycle.[3] The vertical dimension represents the time sequence of clock cycles. An empty (shaded) slot represents an unused execution slot in one pipeline. A no-op is indicated by N.

The first three illustrations in Figure 17.8 show different approaches with a scalar (i.e., single-issue) processor:

- **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading.
- **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.

[3] Issue slots are the position from which instructions can be issued in a given clock cycle. Recall from Chapter 14 that instruction issue is the process of initiating instruction execution in the processor's functional units. This occurs when an instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

symmetric multithreading. That is, the processor has a superscalar architecture and can issue instructions from one or both threads in parallel. At the end of the pipeline, separate thread resources are needed to commit the instructions.

17.5 CLUSTERS

An important and relatively recent development computer system design is clustering. Clustering is an alternative to symmetric multiprocessing as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define 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. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

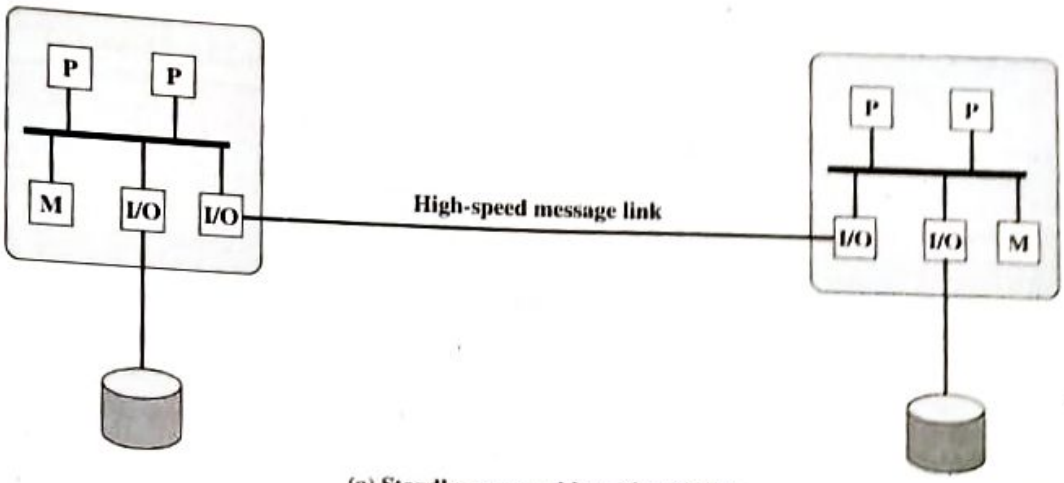
[BREW97] lists four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

- **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.

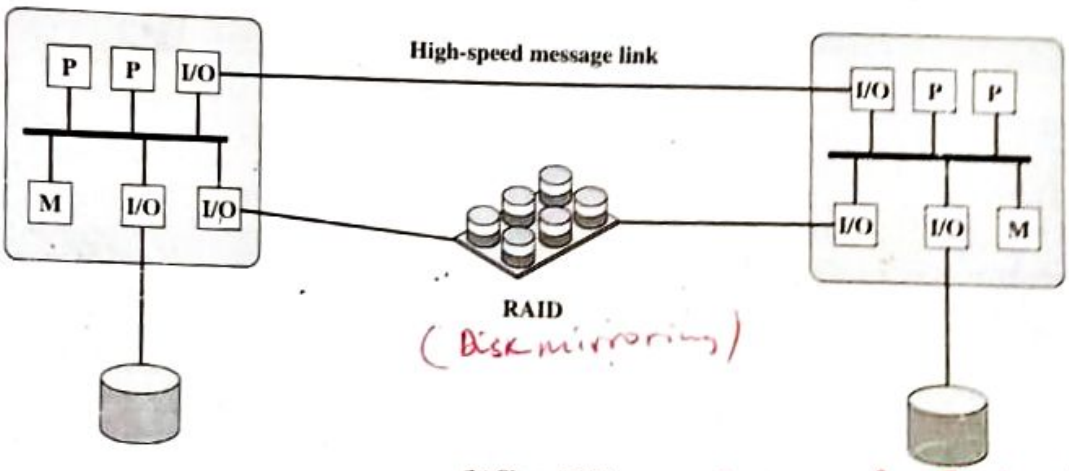
Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 17.10a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so that there is a connection between the server cluster and remote client systems. Note that in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 17.10, the other alternative is a shared-disk cluster. In this case, there generally is still a message link between



(a) Standby server with no shared disk



(b) Shared Disk

Figure 17.10 Cluster Configurations

nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In this figure, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so that the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.

A clearer picture of the range of cluster options can be gained by looking at functional alternatives. Table 17.2 provides a useful classification along functional lines, which we now discuss.

A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and

volumes on all of the disks. This approach requires the use of some type of locking facility to ensure that data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

FAILURE MANAGEMENT How failures are managed by a cluster depends on the clustering method used (Table 17.2). In general, two approaches can be taken to dealing with failures: 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 In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **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

Figure 17.11 shows a typical cluster 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. [HWAN99] lists the following as 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.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single system image.

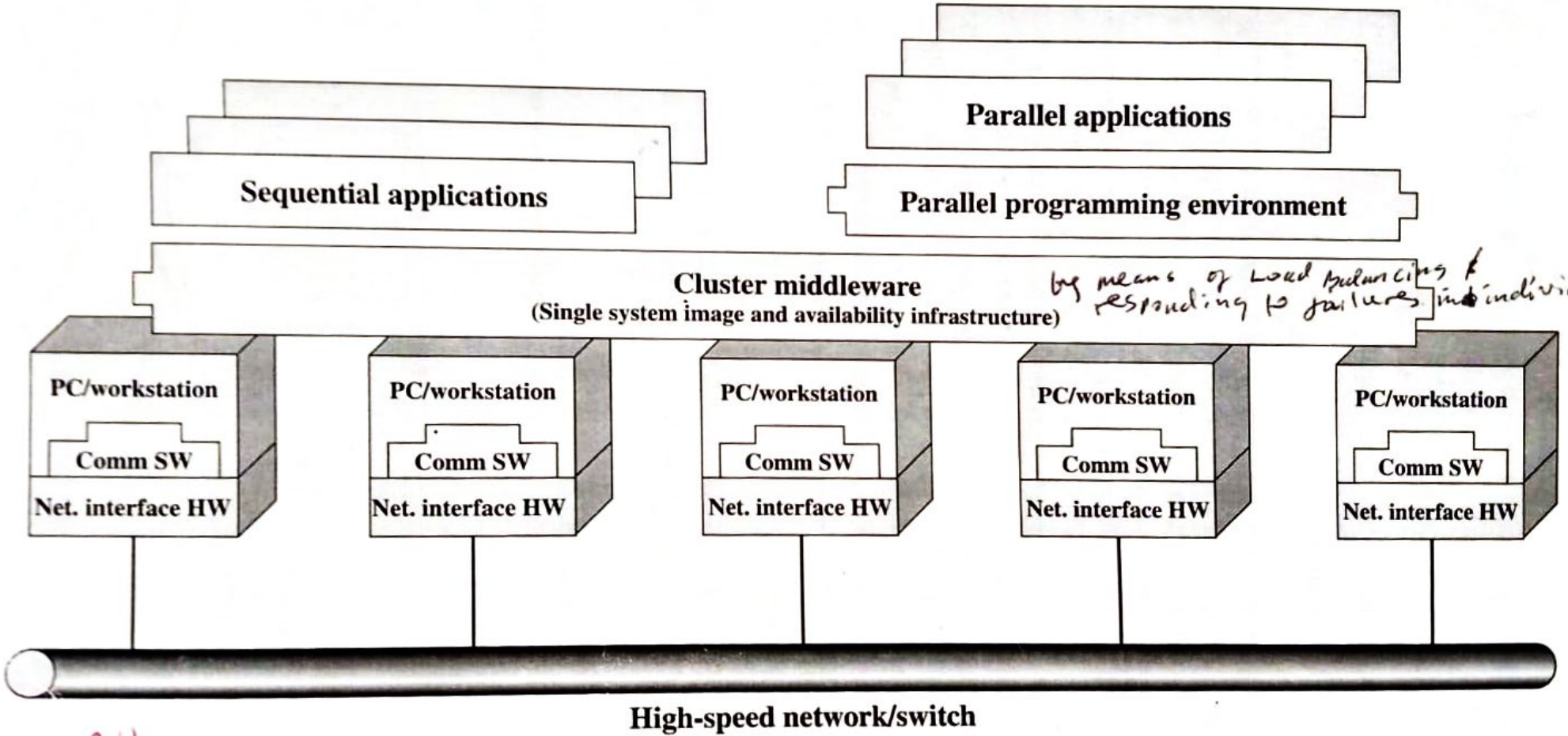


Figure 3.14 Cluster Computer Architecture [BUYY99a]

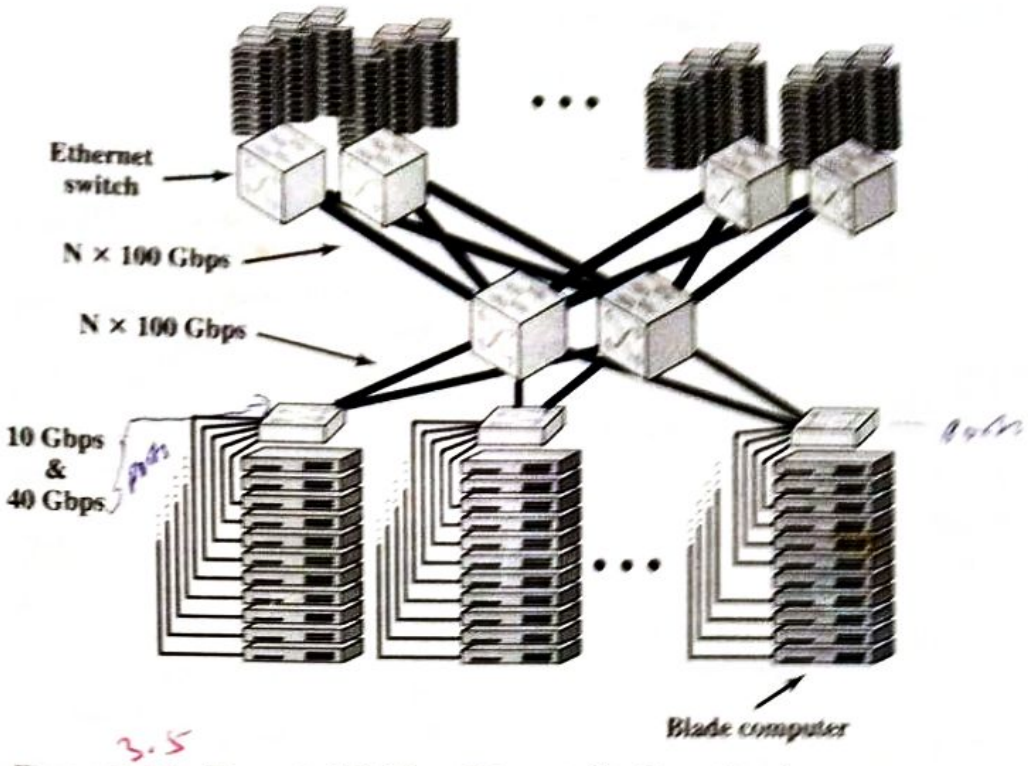


Figure 17.12 Example 100-Gbps Ethernet Configuration for Massive Blade Server Site

Returning to Figure 17.11, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

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.

An example of the application is shown in Figure 17.12, taken from [NOWE07]. The trend at large data centers, with substantial banks of blade servers, is the deployment of 10-Gbps ports on individual servers to handle the massive multimedia traffic provided by these servers. Such arrangements are stressing the on-site Ethernet switches needed to interconnect large numbers of servers. A 100-Gbps rate provides the bandwidth required to handle the increased traffic load. The 100-Gbps Ethernet switches are deployed in switch uplinks inside the data center as well as providing interbuilding, intercampus, wide area connections for enterprise networks.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP schemes have been around far longer.

The main strength of the SMP approach is that 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.

Over the long run, however, 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.

17.6 NONUNIFORM MEMORY ACCESS

In terms of commercial products, the two common approaches to providing a multiple-processor system to support applications are SMPs and clusters. For some years, another approach, known as nonuniform memory access (NUMA), has been the subject of research and commercial NUMA products are now available.

Before proceeding, we should define some terms often found in the NUMA literature.

- **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. The SMP organization discussed in Sections 17.2 and 17.3 is UMA.
- **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. The commercial products that have received much attention recently are CC-NUMA systems, which are quite distinct from both SMPs and clusters. Usually, but unfortunately not always, such systems are in fact referred to in the commercial literature as CC-NUMA systems. This section is concerned only with CC-NUMA systems.

Motivation

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. For example, Silicon Graphics' Power Challenge SMP is limited to 64 R10000 processors in a single system; beyond this number performance degrades substantially.

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. For example, the Silicon Graphics Origin NUMA system is designed to support up to 1024 MIPS R10000 processors [WHIT97] and the Sequent NUMA-Q system is designed to support up to 252 Pentium II processors [LOVE96].

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

Figure 17.13 depicts a typical CC-NUMA 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. For example, each Silicon Graphics Origin node includes two MIPS R10000 processors; each Sequent NUMA-Q node includes four Pentium II processors. 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. Although implementations differ as to details, in general terms we can say that 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. To see how this scheme works, we give an example taken from [PFIS98]. 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.

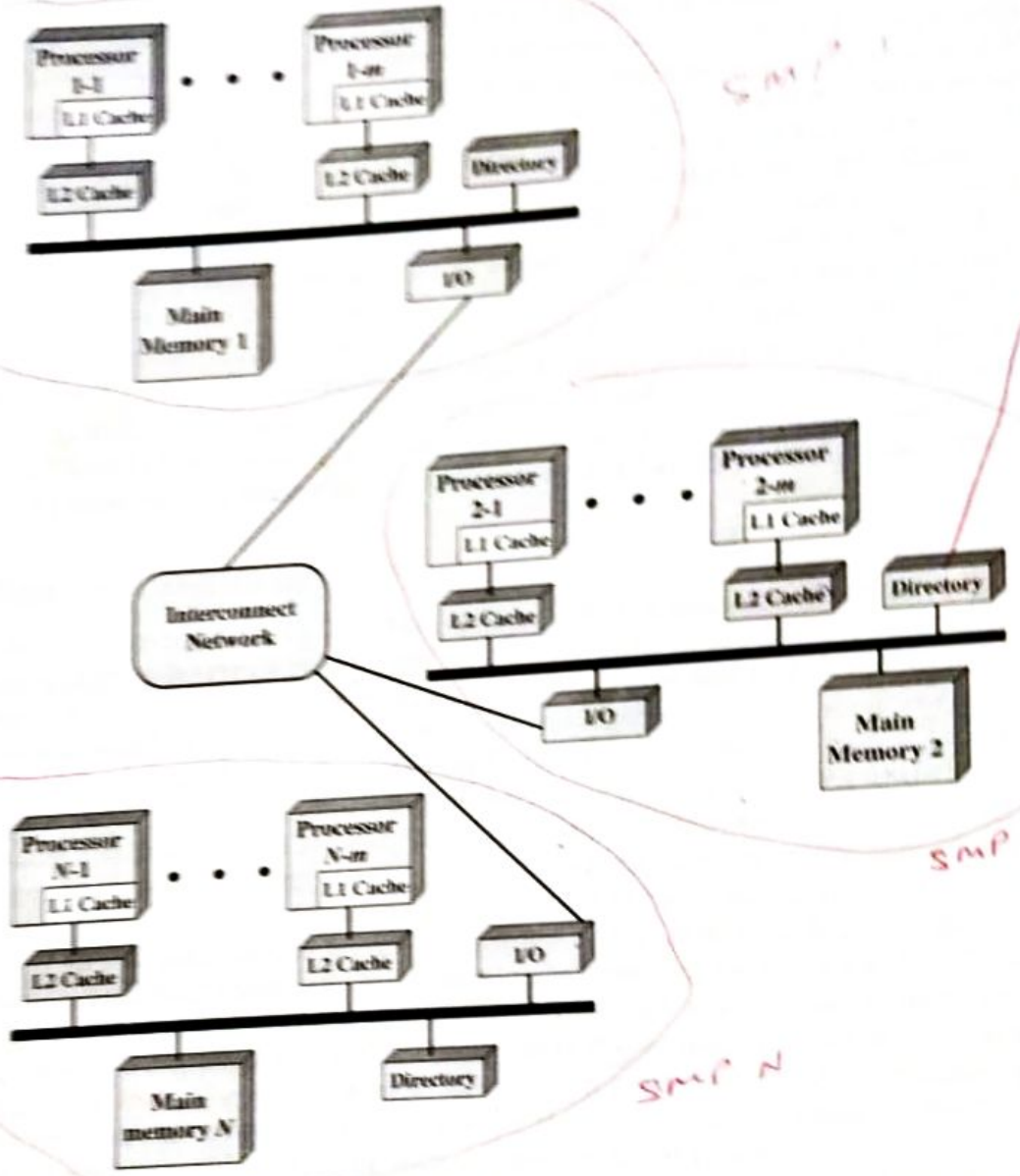


Figure 17.13 CC-NUMA Organization

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.

The preceding sequence explains how data are read from a remote memory using hardware mechanisms that make the transaction transparent to the processor. On top of these mechanisms, some form of cache coherence protocol is needed. Various systems differ on exactly how this is done. We make only a few general remarks here. First, 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.

NUMA Pros and Cons

The main advantage of a CC-NUMA system is that it 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. There is reason to believe that this performance breakdown can be avoided. 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. The Sequent designers report that such spatial locality does appear in representative applications [LOVE96]. 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; the Silicon Graphics designers report success with this approach [WHIT97].

Even if the performance breakdown due to remote access is addressed, there are two other disadvantages for the CC-NUMA approach. Two in particular are discussed in detail in [PFIS98]. 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, already mentioned, 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; the interested reader is referred to [PFIS98].

Vector Processor Simulator

Table 17.2 Clustering Methods: Benefits and Limitations

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.
(i) 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.
(ii) 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.
(iii) 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.

if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term *cluster* is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (Figure 17.10a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so that the load is balanced and high utilization is achieved. It is desirable to have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete the application. For this to happen, data must constantly be copied among systems so that each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (Figure 17.10b). In one variation on this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so that some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so that each computer has access to all of the