

# MCA 503 Distributed Operating Systems and Parallel Computing

Unit # 1

## Introduction to Distributed Systems

D M Patel  
lecturer  
GDCST

# Introduction

- Computer systems are undergoing a revolution.
- From 1945, when the modern computer era began, until about 1985, computers were large and expensive.
- Even minicomputers normally cost tens of thousands of dollars each.
- As a result most organizations had only a handful of computers. And for lack of a way to connect them these operated independently from one another.
- After in the mid 1980s, two advances in the technology began to change that situation.
- First was the development of powerful microprocessors.

# Introduction

- Even many of these had the computing power of a decent sized mainframe computer, but for the fraction of the price.
- The amount of improvement is enormous.
- From a machine that cost 10 million dollars and executed 1 instruction per second, we have come to machines that cost 1000 dollars and execute 10 million instructions per second, a price/performance gain  $10^{11}$ .
- The second development was the invention of high-speed compute networks. The local area network or LAN allows dozens, or even hundreds, of machines within a building to be connected in such a way that

# Introduction

Small amounts of information can be transferred between machines in a milliseconds or so.

➤Larger amounts of data can be moved between machines at rates of 10 to 100 million bits/sec and sometimes more.

➤The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of CPUs connected by a high speed network. They are usually called distributed systems, In contrast to the previous centralized system or single processor systems consisting of a single CPU, its memory, peripherals and

# Introduction

Some terminals.

- Distributed systems need radically different software than centralized system do. In particular the necessary operating systems are only beginning to emerge.
- The first few steps have been taken but there is still a long way to go.
- What is a distributed system?
- For our purpose distributed system is a collection of independent computers that appear to the users of the system as a single computer.

# Introduction

- This definition has two aspects.
- The first one deal with the hardware: the machines are autonomous.
- The second one deal with the software: the users think of the system as a single computer.
- So for distributed system both are essential.
- Example of the distributed systems.
  - OLTP (Online Transaction Processing)  
Online reservation systems in traveling agencies are an example of OLDP. In these kind of systems there is a central system and there are many other machines that are connected to the central system from different geographical locations.

# Introduction

- Banking Applications
- Database Management Applications

In the above systems there are usually clients and servers and there is network in between. On the server side there may be databases and database management systems and on the client side there may be any number of clients with different applications. Client applications may be thick or thin. Thin client programs have very little business logic in them like an applet or an HTML browser but thick client programs have a lot of business logic and computations in them like banking or simulation applications.

- Goals: Just because it is possible to build the distributed system does not mean a good idea. Actually there are some advantages of distributed system over centralized system.

# Introduction

- Advantages of distributed system over Centralized system.
- The real driving force behind the trend toward decentralization is **economics**. The computing power of the CPU is proportional to the square of its price. By paying twice as much, u can get four times the performance. This observation fit the mainframe technology of its time.
- With microprocessor technology this observation no longer holds. For a few hundred dollars u can get a CPU chip that can execute more instructions per second than one of the largest 1980's mainframes.



# Introduction

And if u r willing to pay twice as much u get the same CPU but running at a somewhat higher clock speed. As a result the most cost – effective solution is frequently to harness a large number of cheap CPUs together in a system. Thus distributed systems have potentially much **better price/performance ratio** than a single large centralized system.

- Actually not only better price/performance ratio than a single mainframe, but may yield an absolute performance that no mainframe can even achieve.
- Next reason is that some applications are inherently distributed.
- Another potential advantage of a distributed system over a centralized system is higher reliability. By distributing the workload over many machines a single chip failure will bring down at most one machine, leaving the rest intact.

If 5 percent of the machines are down at the moment, the system should be able to continue to work a 5 percent loss in performance.

➤ Finally, incremental growth is also potentially a big plus. Suppose one company deploys one mainframe to perform all its task, now suppose its workload gets increased so company has to buy new mainframe which is certainly not a feasible solution, but if the case is distributed one then suppose 5 percent workload has been increased then we can certainly deploy enough computers or resources to fulfill required workload.

# Advantages of DS over centralized system

<b>Economics</b>	<b>Microprocessors offer a better price/performance than mainframe</b>
<b>Speed</b>	<b>A distributed system may have more total computing power than mainframe</b>
<b>Inherent distribution</b>	<b>Some application involve spatially separated machines</b>
<b>Reliability</b>	<b>If one machine crashes, the system as a whole can still survive</b>
<b>Incremental growth</b>	<b>Computing power can be added in small increment</b>

# Advantages of DS over Independent PCs

<b>Data sharing</b>	<b>Allow many users access to a common data base</b>
<b>Device sharing</b>	<b>Allow many users to share expensive peripherals like color printers</b>
<b>Communication</b>	<b>Make human to human communication easier,</b>
<b>Flexibility</b>	<b>Spread the workload over the available machines in the most cost effective way.</b>

# Disadvantages of distributed system

<b>Software</b>	<b>Little software exists at present for distributed systems</b>
<b>Networking</b>	<b>The network can saturate or cause other problems</b>
<b>Security</b>	<b>Easy access also applies to secret data</b>

# Hardware concepts

## *Interconnect*

There are different ways in which we can connect CPUs together. The most widely used classification scheme (**taxonomy**) is **that created by Flynn in 1972. It classifies machines** by the number of instruction streams and the number of data streams. An instruction stream refers to the sequence of instructions that the computer processes. Multiple instruction streams means that different instructions can be executed concurrently. Data streams refer to memory operations. Four combinations are possible:

➤ **SISD** Single instruction stream, single data stream. This is the traditional uniprocessor computer.

➤ **SIMD** Single instruction stream, multiple data streams. This is an array processor; a single instruction operates on many data units in parallel.

➤ **MISD** Having multiple concurrent instructions operating on a single data element makes no sense. This isn't a useful category.

➤ **MIMD** Multiple instruction stream, multiple data streams. This is a broad category covering all forms of machines that contain multiple computers, each with a program counter, program, and data. It covers parallel and distributed systems.

➤ Since the MIMD category is of particular interest to us, we can divide it into further classifications.


Note:

➤ We refer to machines **with shared memory** as **multiprocessors** and to machines **without shared memory** as **multicomputers**.

➤ Multiprocessor's aim is to get speed up on a single problem, means all the CPUs work on a single problem they are also known as a parallel systems.

➤ Multicomputer is a collection of computers actually not the collection of CPUs like multiprocessor so they work on different problems but tries to appear to its user as a single uniprocessor.





➤ Interconnection of the CPUs can be either BUS based or switched based so now we have four architectures present for interconnecting CPUs.

# MIMD

Parallel and  
distributed computers

Tightly Coupled

Loosely Coupled

Multiprocessors  
(Shared memory)

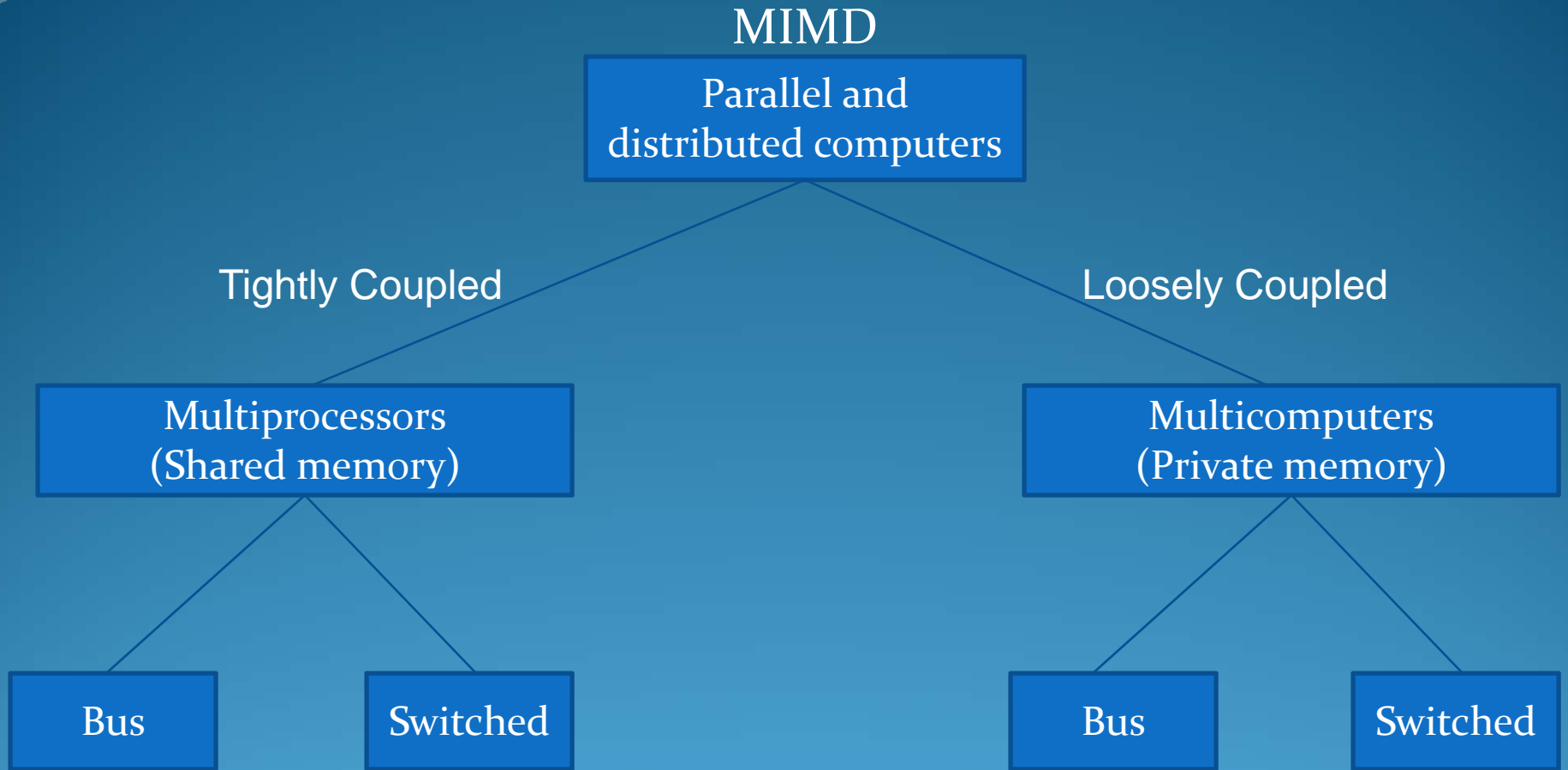
Multicomputers  
(Private memory)

Bus

Switched

Bus

Switched



# Bus-Based Multiprocessors

- Bus based **multiprocessors consist of some number of CPUs all connected to a common bus**, along with a memory module.
- A simple configuration is to have a high-speed backplane or motherboard into which CPU and memory cards can be inserted.
- A typical bus has 32 or 64 address lines, 32 or 64 data lines, and perhaps 32 or more control lines, all of which operate in parallel.
- To read a word of memory, a CPU puts the address of the word it wants on the bus address lines, then puts a signal on the appropriate control lines to indicate that it wants to read.
- The memory responds by putting the value of the word on the data lines to allow the requesting CPU to read it in.

➤ In a bus-based system, all CPUs are connected to one bus (Figure 1). System memory and peripherals are also connected to that bus. If CPU *A* writes a word to memory and CPU *B* can read that word back immediately, so memory **is coherent**.

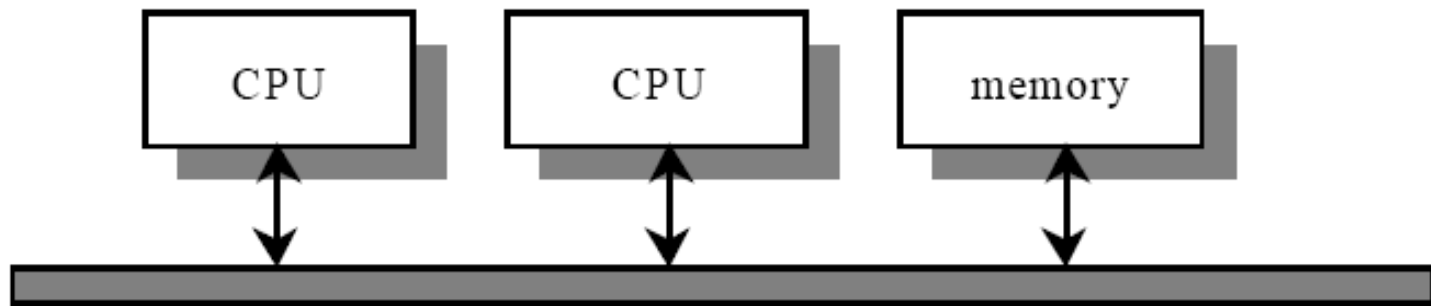
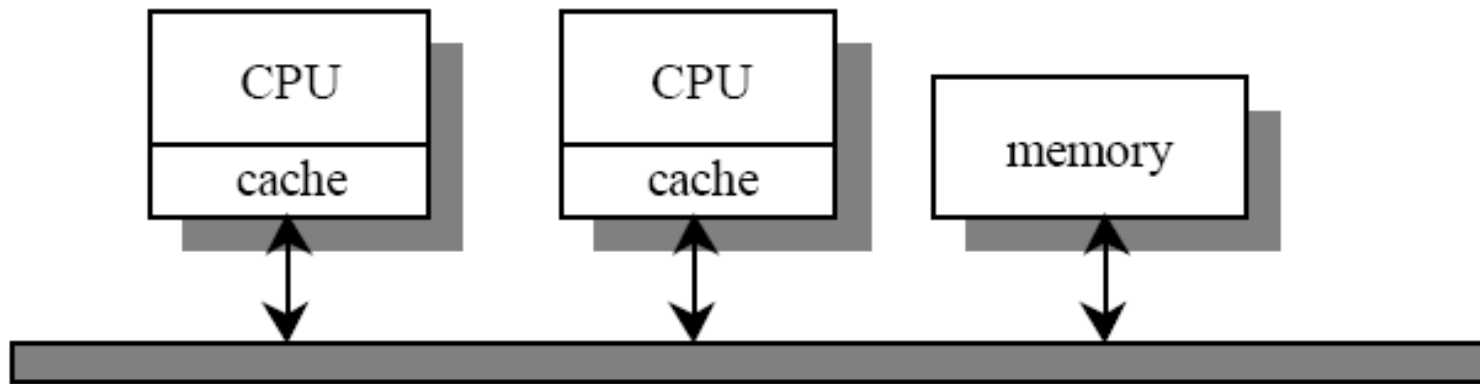


Figure 1. Bus-based interconnect

- A bus can get overloaded rather quickly with each CPU accessing the bus for all data and instructions. So good for only small numbers of CPUs, but creates problem with larger numbers of CPUs.
- A solution to this is to add **cache memory between the CPU and the bus** (Figure 2).



➤ The cache holds the most recently accessed regions of memory. This way, the CPU only has to go out to the bus to access main memory only when the regions are not in its cache.

➤ The problem that arises now is that if two CPUs access the same word (or same region of memory) they load it into their respective caches and make future references from their cache.

➤ Suppose CPU A modifies a memory location. The modification is local to its cache so when CPU B reads that memory location, it will not get A's modification.

➤ One solution to this is to use a **write-through cache**. In this case, **any write is written not only to the cache, but also sent on the bus to main memory**. Writes generate bus traffic now, but reads generate it only if the data needed is not cached. We expect systems to have far more reads than writes.

➤ **In this design cache hits for reads do not cause bus traffic, but cache misses for reads, and all writes, hits and misses, cause bus traffic.**

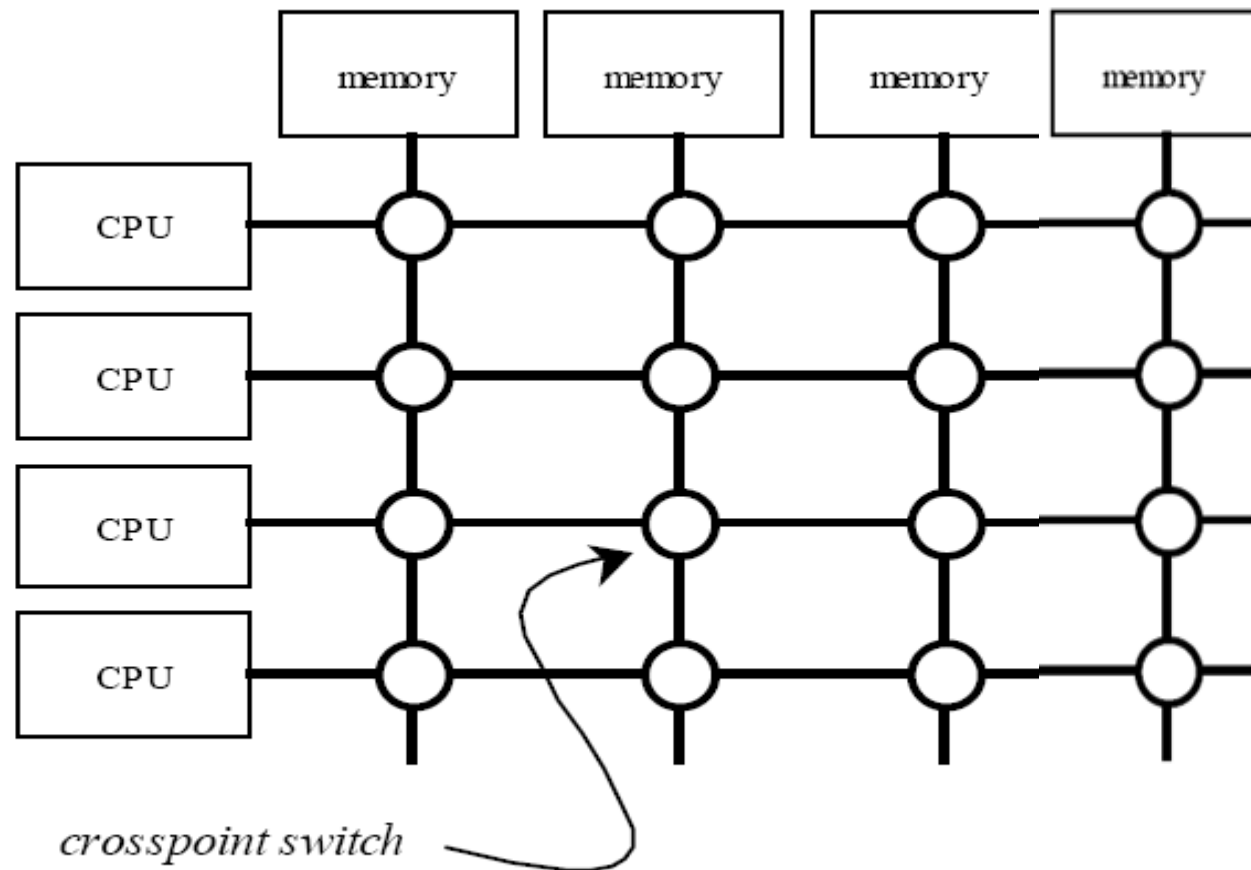
- Hit for read -> does not create traffic
- Miss for read -> does create traffic
- Hit for write -> does create traffic
- Miss for write -> does create traffic

➤ This alone is not sufficient, since other CPU caches may still store local copies of data that has now been modified. We can solve this by having every cache monitor the bus. If a cache sees a write to a memory location that it has cached, it either removes the entry in its cache (invalidates it) or updates it with the new data that's on the bus<sup>1</sup>. If it ever needs that region of memory again, it will have to load it from main memory. This is known as a **snoopy cache** (because it *snoops on the bus*).



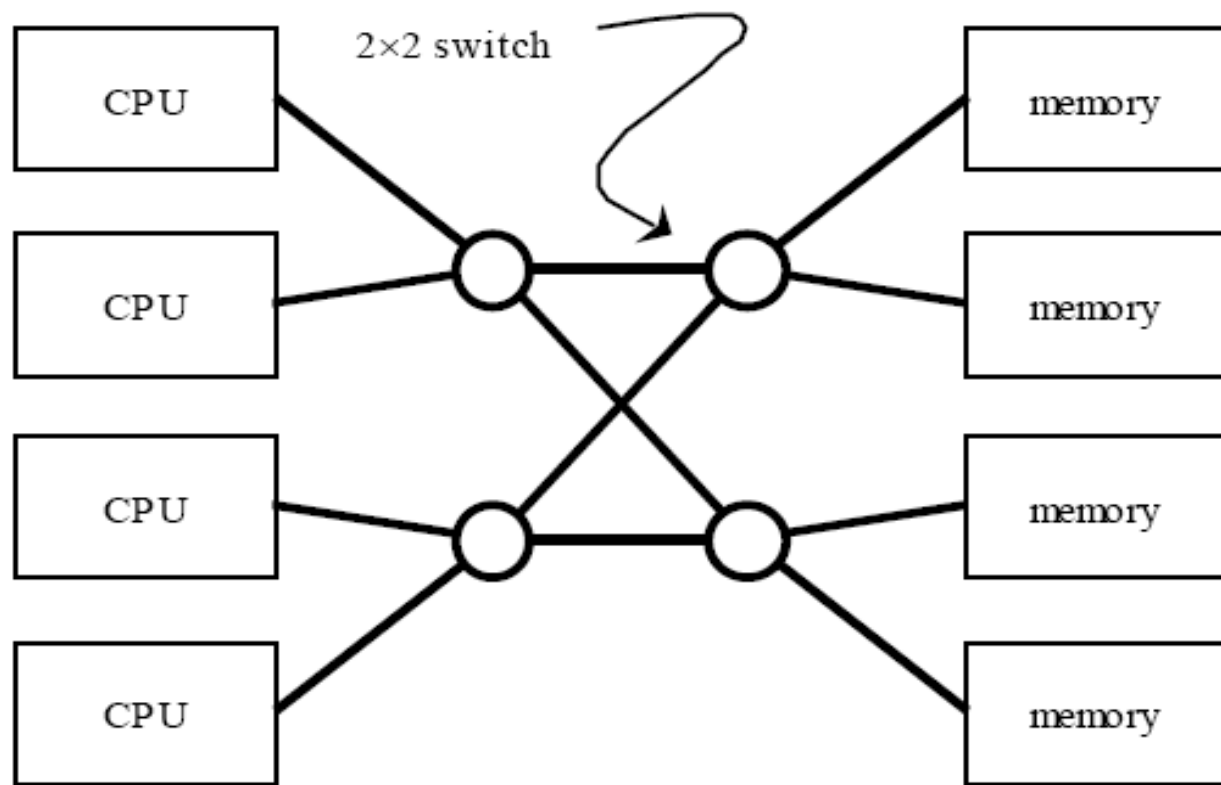
# Switched multiprocessors

- To build a multiprocessor with more than 64 processors, a different method is needed to connect the CPUs with the memory.
- A bus-based architecture doesn't scale to a large number of CPUs (e.g. 64).
- Using switches enables us to achieve a far greater CPU density in multiprocessor systems.
- An  $m \times n$  **crossbar switch** is a switch that allows any of  $m$  *elements* to be switched to any of  $n$  *elements*.
- A crossbar switch contains a *crosspoint switch* at each switching point in the  $m \times n$  array, so  $m \times n$  crosspoint switches are needed (Figure 3).



**Figure 3. Crossbar interconnect**

- To use a crossbar switch, we place the CPUs on one axis (e.g.  $m$ ) and the *break the memory into a number of chunks which are placed on the second axis* (e.g.  $n$  memory chunks). There will be a delay only when multiple CPUs try to access the same memory group.
- A problem with crossbar switches is that they are expensive: to connect  $n$  CPUs with  $n$  memory modules *requires  $n^2$  crosspoint switches*.
- We'd like an alternative to using this many switches. To reduce the number of switches and maintain the same connectivity requires increasing the number of switching stages.
- This results in an **omega network** (Figure 4), which, for a system of  $n$  CPUs and  $n$  memory modules, *requires  $\log n$  (base 2) switching stages, each with  $n/2$  switches for a total of  $(n \log n)/2$  switches*.



**Figure 4. Omega interconnect**

- This network contains four 2 x 2 switches, each having two inputs and two outputs. Each switch can route either input to either output.
- A careful look at the figure will show that with proper settings of the switches, every CPU can access every memory.

- **This is better than  $n^2$  but can still amount to many switches.** As we add more switching stages, we find that our delay increases. With 1024 CPUs and memories, we have to pass through ten switching stages to get to the memory and through ten to get back.
- To try to avoid these delays, we can use a hierarchical memory access system: each CPU can access its own memory quickly but accessing other CPU's memory takes longer. This is known as a **Non-Uniform Memory Access, or NUMA, architecture.**
- It provides better average access time but placement of code and data to optimize performance becomes difficult.

# Summary for multiprocessors

- To summarize, bus based multiprocessors, even with snoopy caches, are limited by the amount of bus capacity to about 64 CPUs at most.
- To go beyond that requires a switching network, such as a crossbar switch, an omega switching network, or something similar.
- Large crossbar switches are very expensive, and large omega networks are both expensive and slow.
- NUMA machines requires complex algorithms for good software placement.
- The conclusion is clear: building a large, tightly coupled, shared memory multiprocessor is possible, but is difficult and expensive.

# Bus based multicomputers

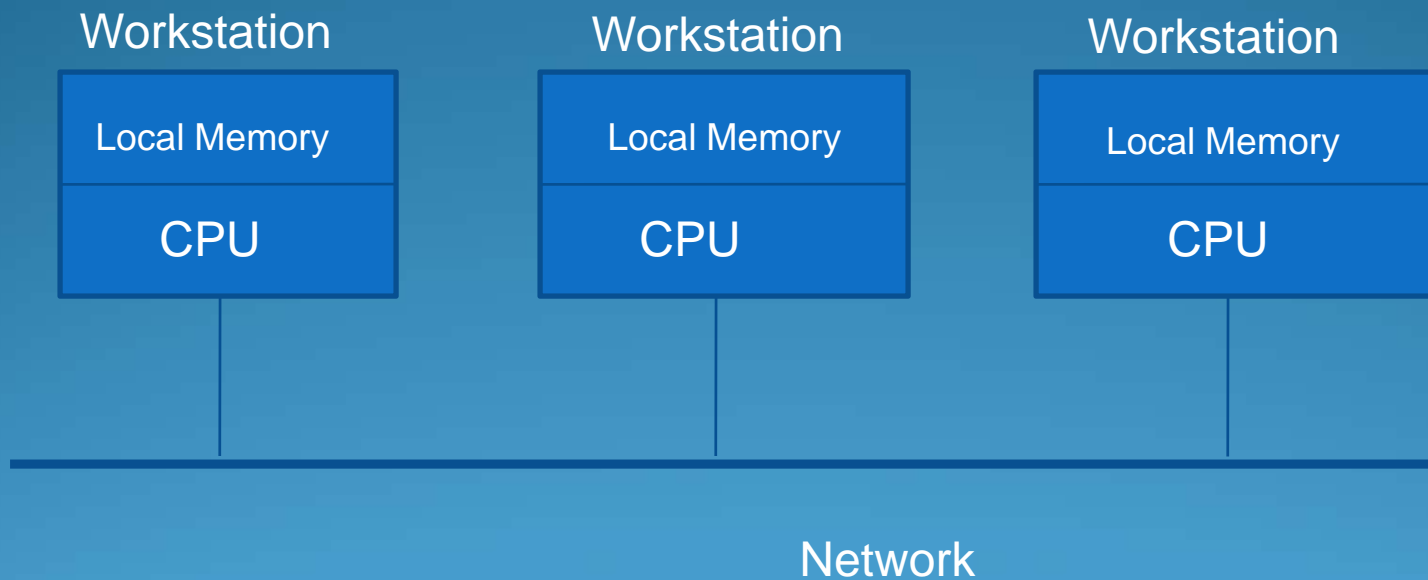
➤ On the other hand, building a multicomputer (no shared memory) is easy.

➤ Each CPU has a direct connection to its own local memory.

Bus-based multicomputers are easier to design in that we don't need to contend with issues of shared memory: every CPU simply has its own local memory. However, without shared memory, some other communication mechanism is needed so that processes can communicate and synchronize as needed.

➤ The communication network between the two is a bus (for example, an Ethernet local area network). The traffic requirements are typically far lower than those for memory access (so more systems can be attached to the bus). Next slide shows the figure for bus based multicomputer.





A multicomputer consisting of workstations on a LAN

➤ It looks topologically similar to the bus based multiprocessor, but since there will be much less traffic over it, it need not be a high speed backplane bus. In fact it can be a much lower speed LAN (typically, 10 -100 Mbps, compared to 300 Mbps and up for a backplane bus).

➤ Extra:

➤ A bus can either be a system bus or a local area network. Bus-based multicomputers most commonly manifest themselves as a collection of workstations on a local area network.

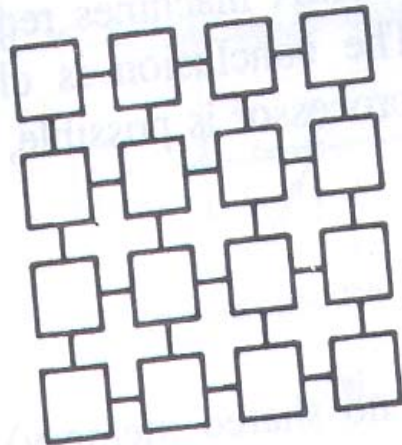
➤ System busses generally have speeds of about 300 Mbps – 1 Gbps. Typical speeds for local area networks are from 10 Mbps to 1 Gbps, with some operating in the Kbps and low megabit per seconds range (for infrared and wireless).

# Switch based multicomputers

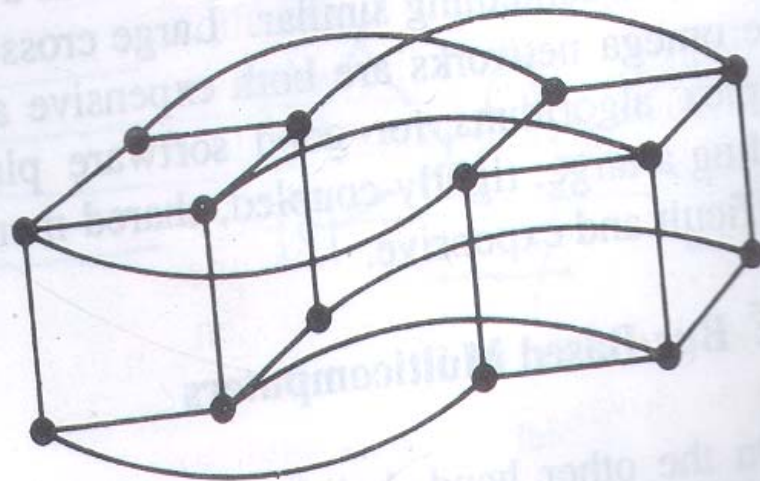
- In a switched multicomputer system, each CPU still has its own local memory but it also has a switched interconnect to its neighbors. Common arrangements are a **grid**, **cube**, or hypercube network. Only nearest neighbors are connected in this network; messages to others require multiple hops.
- Grids are easy to understand and lay out on printed circuit boards. They are best suited to problems that have an inherent two-dimensional nature, such as graph theory or vision.

# Switch based multicomputers

lems that have an inherent  
vision (e.g., robot eyes or analyzing photographs).



(a)



(b)

Fig. 1.8 (a) Grid. (b) Hypercube.

# Switch based multicomputers

- A hypercube is an  $n$ -dimensional cube. The hypercube of fig. 1-8(b) is **four-dimensional**. It can be thought of as two ordinary cubes, each with 8 vertices and 12 edges.
- Each **vertex is a CPU**. Each **edge is a connection between two CPUs**. The corresponding vertices in each of the two cubes are connected.
- To expand the hypercube to **five dimensions**, we should add another **set of two interconnected cubes to the figure**, connect the **corresponding edges in the two halves**, and so on.
- For an  $n$ -dimensional hypercube, each CPU has  $n$  connections to other CPUs.
- Thus the complexity of the wiring increases only logarithmically with the size. Since nearest neighbors are connected, many messages have to make several hops to reach their destination.

# Switch based multicomputers

- However longest possible path also increases logarithmically with the size, in contrast to the grid where it grows as the square root of the number of CPUs.
- Hypercubes with 1024 CPUs have been commercially available for several years and hypercubes with as many as 16384 CPUs are starting to become available.

So far we have seen 4 kinds of hardware architecture and 2 kinds of software (loosely coupled software and tightly coupled s/w)

So theoretically there can be 8 possible combination but only few are legal and popular.

- Loosely Coupled S/w on Loosely Coupled H/w
  - E.g LAN
- Tightly Coupled S/w on Loosely Coupled H/w
  - E.g true distributed system
- Tightly Coupled S/w on Tightly Coupled H/w
  - E.g multiprocessor timesharing system

# True Distributed Systems

- Network operating systems are loosely –coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that such a system consists of numerous computers. Each can run its own operating system and do whatever its owner wants. There is no coordination at all except for the rule that client-server traffic must obey the system's protocols.
- The next evolutionary step beyond this is tightly-coupled software on the same loosely coupled (i.e multicomputer) hardware.
- The goal of such a system is to create the illusion in the minds of users that the entire network of computers is a single time sharing system, rather than a collection of distinct machines.
- Some authors refer to this property as the single system image.



# True Distributed Systems

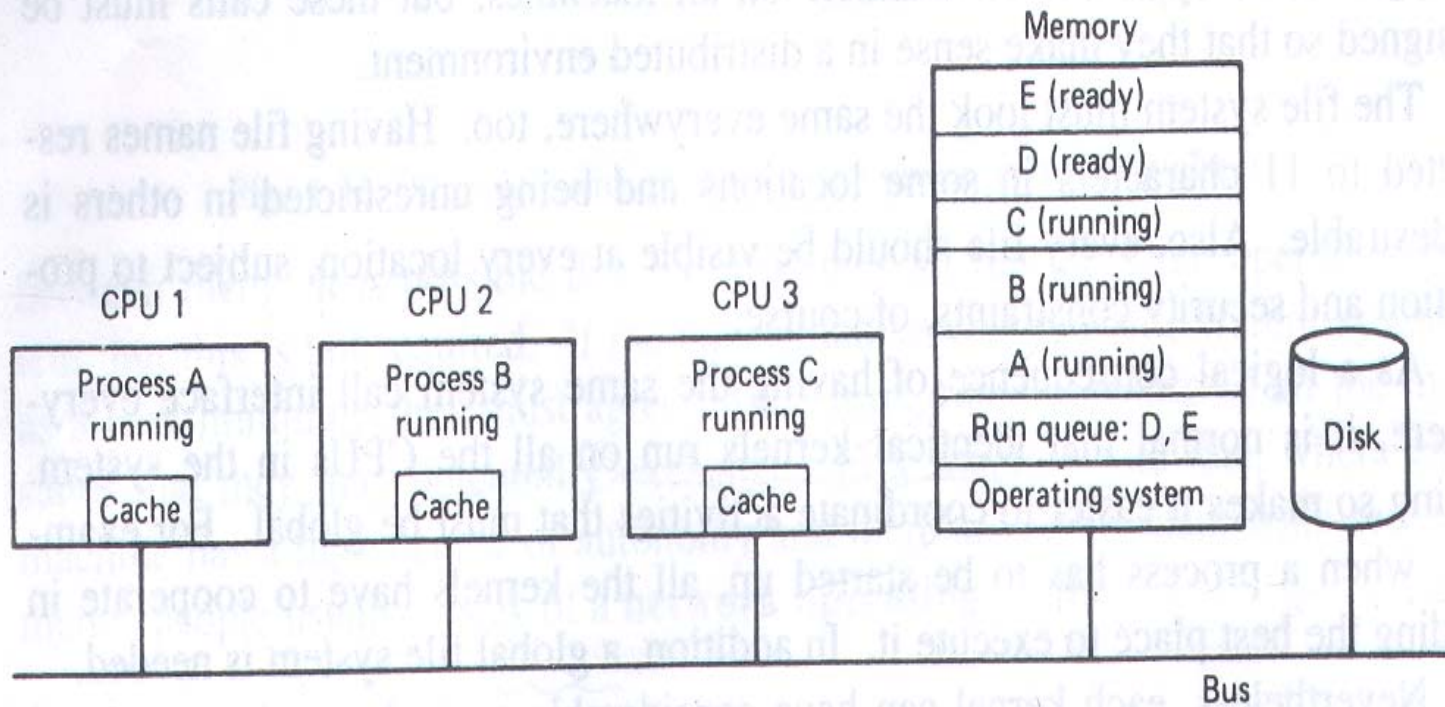
➤ In slightly different way we can say that a distributed system is one that runs on a collection of networked machines but acts like a virtual uniprocessor. No matter how it is expressed, the essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. No current system fulfills this requirements.

➤ some characteristics of true distributed system.

- There should be a single, global, interprocess communication mechanism so that any process can talk to any other process.
- Process management should be same everywhere.
- The file system should look same everywhere.
- Same system call interface should be there everywhere.
- Identical kernel should be run on all CPUs.

# Multiprocessor time sharing system

- The last combination we wish to discuss is tightly coupled software on tightly coupled hardware.
- The key characteristic of this class of system is the existence of a single run queue: a list of all the processes in the system that are logically unblocked and ready to run.
- The run queue is the data structure kept in the shared memory.
- Assume that process B goes for I/O or time quantum for CPUs expires.
- In both cases CPUs would look for the next process from the list to run it.



**Fig. 1-11.** A multiprocessor with a single run queue.

# Difference between three kinds of systems we have examined above.

item	Network operating system	Distributed Operating system	Multiproc essor Operating System
Does it look like a virtual uniprocessor?	No	Yes	Yes
Do all have to run same operating system?	No	Yes	Yes
How many copies of the operating system are there?	N	N	1
How is communication is achieved?	Shared files	Messages	Shared Memory
Are agreed upon network protocol required?	Yes	Yes	No
Is there a single run queue?	No	No	Yes
Does file sharing have will-defined semantics?	No	Yes	Yes

# Design issues

➤ In the preceding sections we have looked at distributed systems and related topics from both hardware and software point of view. Now we will see some of the key design issues that people contemplating building a **distributed operating system must deal with.**

## ➤ transparency

- **Transparency:** Most important issue is how to achieve the single – system image. In other words how do the system designers fool everyone into thinking that the collection of machines is simply an old-fashioned timesharing system?
- A system that realizes this goal is said to be transparent.
- The concept of transparency can be applied to many different aspect of distributed system.

# Location transparency

➤ refers to the fact that in a true distributed system, users can not tell where the hardware and software resources such as CPUs, printers, files, and databases are located. The name of the resource must not secretly encode the location of the resource. Os manes like `machine1:prog.c` or `machine/prog.c` are not acceptable.

# Migration transparency

- Means that resources must be free to move from one location to another without having their names change.

# Replication transparency

- If the distributed system has replication transparency, the operating system is free to make additional copies of files and other resources on its own without the users noticing.
- For understanding replication transparency consider a collection of  $n$  servers logically connected to form a ring. All servers keep the whole directory structure but holds the only subset of the total resources. If user sends request to one of the servers then if it has it will respond otherwise will forward the request to the next server, and this procedure is repeated until the desired resource is found.



# Concurrent transparency

➤ Distributed system have multiple, independent users. What should the system do when two or more users try to access the same resource at the same time. For example what happens if two users try to update the same file at the sometime? If the system is concurrency transparent the user will not notice the existence of other users. One obvious mechanism is lock the resource if some one has started using it.

# Parallelism transparency

➤ In principle a distributed system is supposed to appear to the users as a traditional, uniprocessor timesharing system. What happens if a programmer knows that his distributed system has 1000 CPUs and he wants to use substantial fraction of them for a chess program that evaluates boards in parallel? **The theoretical answer is that together the compiler, runtime system, and operating system should be able to figure out how to take advantage of this potential parallelism without the programmer even knowing it.**

# reliability

- As we know distributed systems are more reliable than any individual system. Suppose that there are four file servers each with a 0.95 chance of being up at any instant, so the probability of all four being down is 0.000006, so the probability of at least one being available is 0.999994, which is better than 0.95
- Let us consider now from another aspects of reliability
  - **Availability point of view**: if availability is more, then reliability is also more. Availability can be enhanced by a design that does not require simultaneous functioning of substantial number of critical component.
  - Secondly we can replicate the hardware and software resources so that they can be more available.
  - But it also creates the problem of data inconsistency.
  - **Reliability from security point of view**. Files and other resources must be protected from the unauthorized access.

# performance

- Various performance matrices can be used to measure the performance of a distributed system
  - E.g response time, throughput (#jobs/unit time), system utilization, amount of network capacity consumed.
- The performance problem is compounded by the fact that communication which is very essential in distributed system is typically quite slow.
- Sending messages and getting a reply over a LAN takes about 1 msec.
- Most of the time is spent in unavoidable protocol handling at both the ends.
- To optimize performance it is necessary to minimize number of messages.

➤ One possible way is to pay considerable attention to the grain size.

Fine grained parallelism	Coarse grained parallelism
-> jobs that involve large number of small computations, especially ones that interact highly with one another are said to exhibit fine grained parallelism	Jobs that involve large computation rates, and little data, are said to exhibit coarse grained parallelism
-> leads to poor performance of a distributed system	Leads to improvement in the performance of the system.

# scalability

- Potential bottleneck that designers should try to avoid in very large distributed systems
  - Centralized components
  - Centralized tables
  - Centralized algorithms.
- These three component should be minimized if we want to scale the DS for larger number of CPUs.

# flexibility

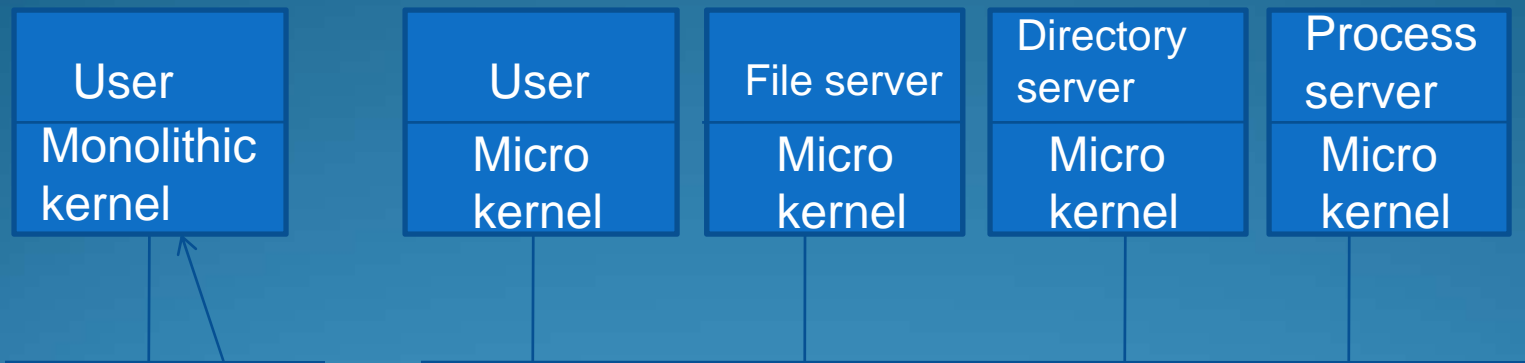
➤ Two approaches regarding the structure of a distributed system

Monolithic kernel	Micro kernel
1. Each machine runs a traditional kernel that provides most services itself.	1. Kernel provides selected (very few) services (as little as possible)
2. Is basically today's centralized operating system augmented with networking facilities.	2. Bulk of operating system services are provided through user level servers.
3. Most system calls are made by trapping to the kernel, having the work performed there and having the result returned to user process.	3. This one is upcoming and new approach

## Micro kernel approach is

- Is flexible
- Modular approach
- It is easy to implement, install and debug new services
- Does not require stopping the system and rebooting a new kernel





Includes file, directory  
and process  
management

network

Monolithic kernel

Micro kernel

