Module 1

* parallel computing

parallel computing is a type of computation in which multiple processor (or cores) execute multiple task simultaneously by dividing large problem into smaller sub-problem.

* classification of parallel computers

Flynn's Taxonomy (Based on instruction streams and data streams)

- SISD (Single instruction streams, Single data stream)
  - ↳ classical von neuman system
  - ↳ Executes one instruction at a time
  - ↳ computes one data value at a time
- SIMD (Single instruction stream, multiple data streams)
  - ↳ multiple data processed simultaneously under one instruction stream.
- MIMD (multiple instruction streams, multiple Data streams)
  - ↳ supports multiple instruction streams
  - ↳ can manage multiple data streams simultaneously.

Memory access classification

① Shared memory system
- cores share access to a common memory
- coordination happens through modifying shared memory locations.

② Distributed memory system
- each core has private memory
- coordination occurs via communication across a network

* SIMD System

Single instruction multiple data.
- it is a key parallel computing architecture described in Flynn's Taxonomy.
- the core idea of SIMD is to execute a single instruction across multiple data elements simultaneously.
- this makes SIMD highly efficient for data-parallel workloads, where the same operation is applied repeatedly across large datasets.
- A typical SIMD System consist of single control unit and multiple datapaths
- the instructions is broadcast from the control unit to the datapaths.
- each datapaths executes the same instruction on its own assigned data.

for example: consider vector addition

```
for (int i=0; i<n; i++)
    x[i] += y[i];
```

If the SIMD system has exactly n datapaths, then during a single instruction cycle, each datapath handles one element addition:

ith datapath loads x[i] and y[i]
performs x[i] += y[i].
stores result back in x[i]

- If the system has m datapaths and m < n, we can simply execute the additions in blocks of m elements at a time

m = 4 n = 15

first add elements 0 to 3, 4 to 7, 8 to 11 & finally elements 12 to 14, in last group of elements - 12 to 14, we are only operating on 3 elements x & y, so one of the four datapath will be idle.

* The requirement that all the datapaths execute the same instruction (or) idle can seriously degrade the overall performance of a SIMD system. for example, suppose we only want to carry out the addition if y[i] is positive:

```
for (i=0; i<n; i++)
    if (y[i] > 0.0)
        x[i] += y[i];
```

- In this setting we must load each element of y into a datapath and determine whether it's positive.
- if y[i] is positive, we can proceed to carry out the addition
- otherwise, the datapath storing y[i] will be idle while the other datapaths carry out the addition.
- Synchronous operation:
  - Datapath must operate synchronously
  - each must wait for the next instruction to be broadcast
  - Datapaths have no instruction storage and cannot delay execution.
- SIMD system are ideal for parallelizing simple loops operating on large arrays of data.
- data parallelism
  - parallelism is achieved by dividing data among processors
  - processors apply the same instructions to their subsets of data.
- SIMD systems often do not perform well on other types of parallel problems aside from data-parallel tasks.

* MIMD System

- MIMD System support multiple simultaneous instruction streams operating on multiple dat streams.
- MIMD systems typically consist of a collection of fully independent processing units (or) cores.
- each core has its own control unit & datapath.
- unlike SIMD System, MIMD system are usually "asynchronous."
- processors can operate at their own pace.
- In many cases, there is no relation.
- unless synchronisation is explicitly imposed by the programmer. processors may execute different statements even if running the same instruction sequences

2 types of MIMD System

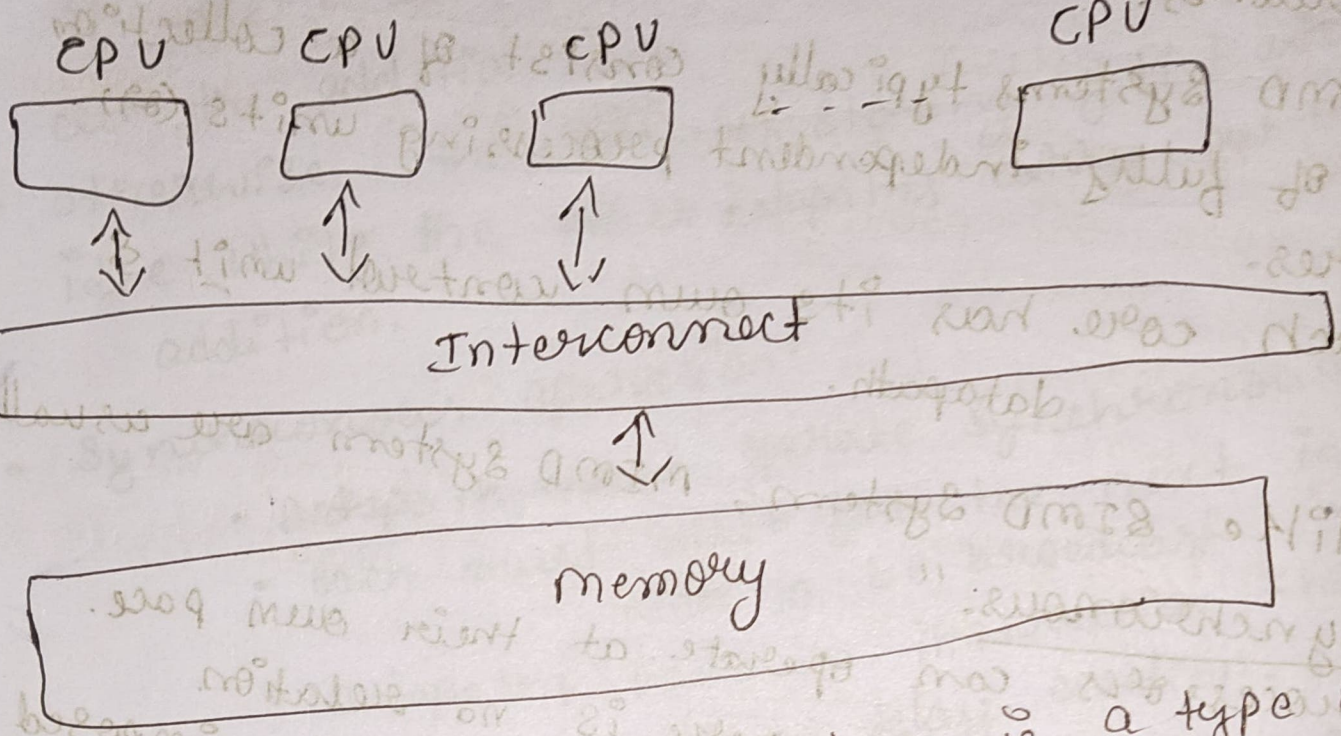
① Shared - memory system

② distributed - memory System

① Shared - memory System

- Autonomous processors are connected to a memory System via an interconnection network
- each processor can acces every memory location

- processors usually communicate implicitly by accessing shared data structure

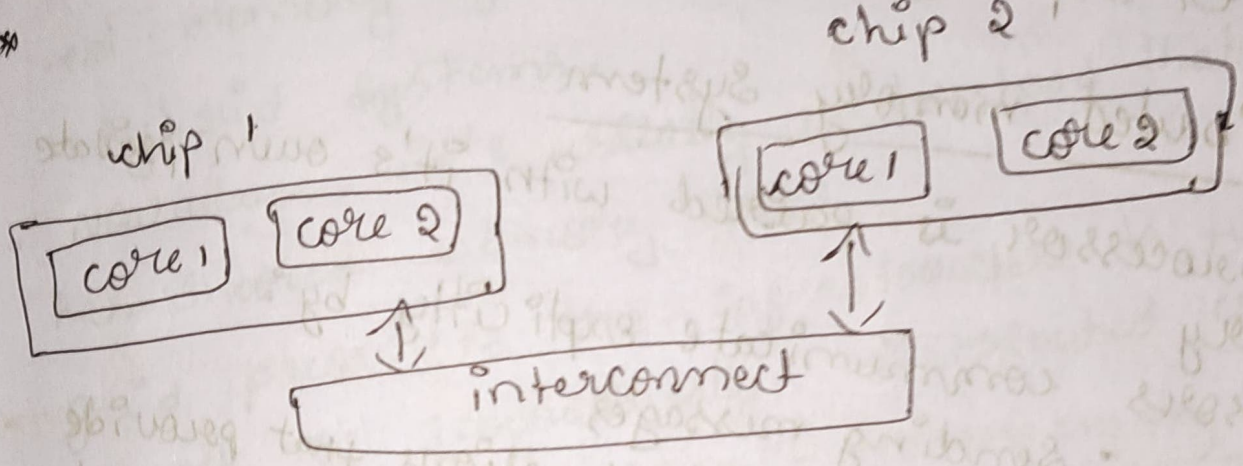


* A shared memory system is a type of computer architecture where multiple processors (or cores) access a common memory space
* It is "widely used in multicore processors and SMP (Symmetric multiprocessing) system
* The most widely available shared-memory system use one (or) more multicore processors
* the cores have private level 1 caches, while others caches may (or) may not be shared between the cores.
* In shared-memory systems with multiple multicore processors, the intercon

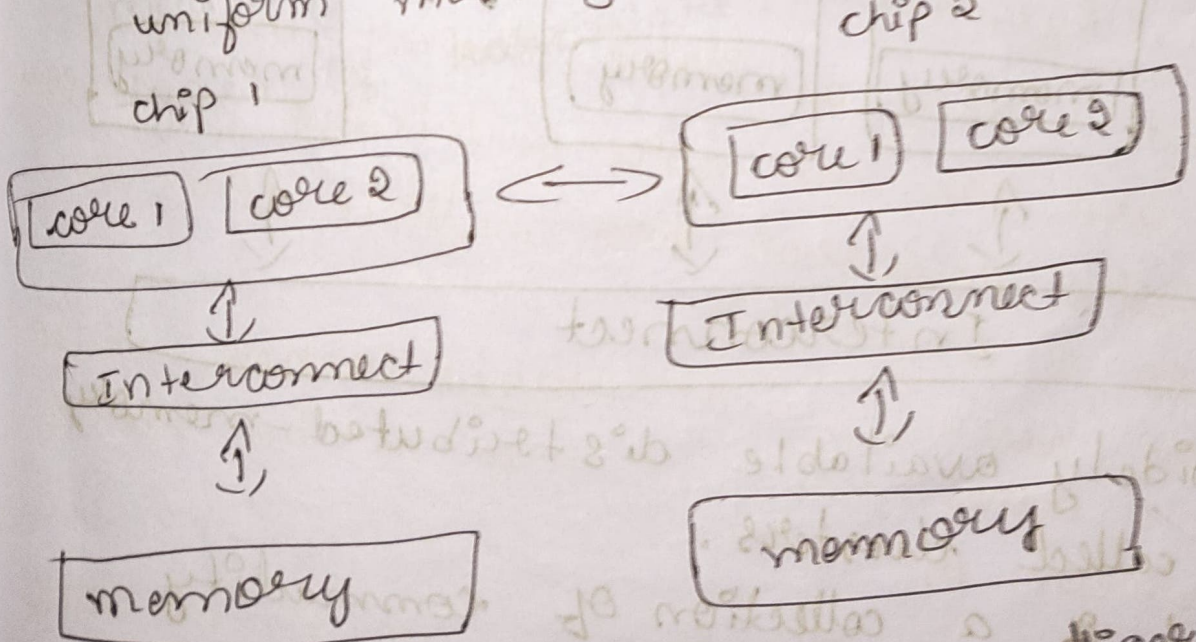
can either connect all the processors directly to main memory, or each processor can have a direct connection to a block of main memory.

* & the processors can access each other's blocks of main memory through special hardware built into the processors.

*



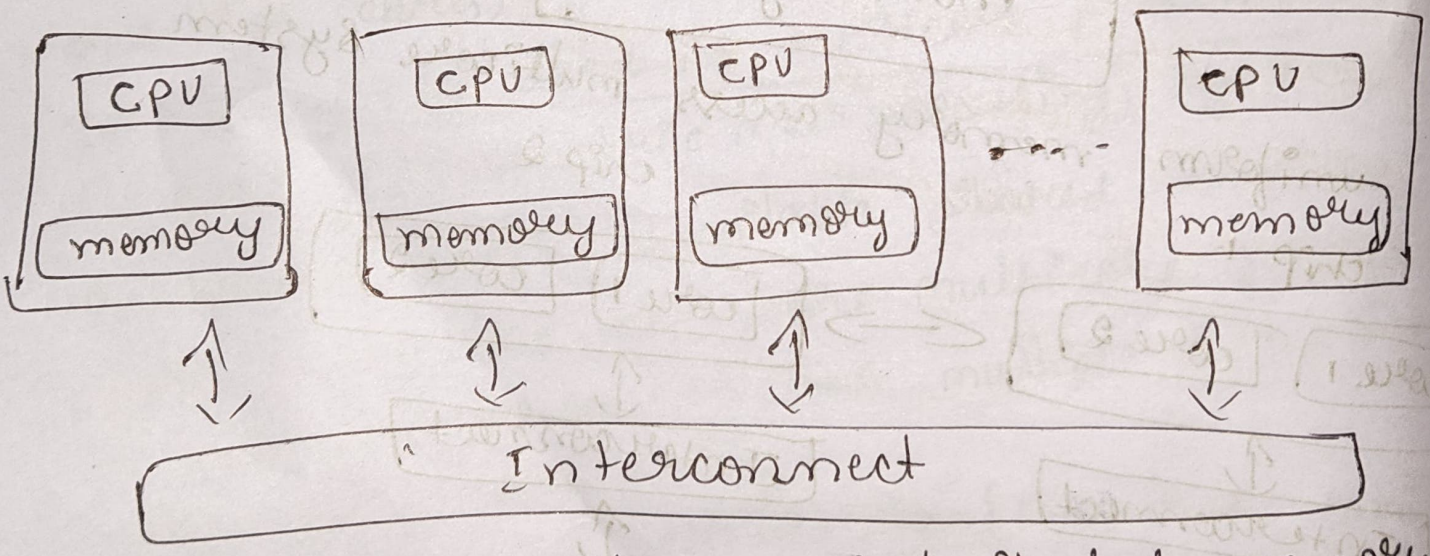
uniform memory access multicore system



non-uniform memory access multicore system

* the first type of system, the time to access all the memory locations will be the same for all the cores. (it is called UMA)
* 2nd type of system. a memory location which a core is directly connected, can be accessed more quickly than a memory location that must be accessed through another chip. (NUMA).

* Distributed memory system
  - each processor is paired with it's own private memory
  - processors communicate explicitly by:
    - sending messages
    - using special functions that provide access to another processor's memory



  - most widely available distributed-memory system called clusters.
  - composed of a collection of commadity systems. (eg. PCs)

- connected by a commodity interconnection network. (eg. Ethernet)
- the nodes of these system, the individual computational units joined together by the communication network - are usualy shared memory system with one (or) more multicore processors.
- To distinguish such systems from pure distributed - memory systems, they are sometimes called hybrid systems.
- Nowadays, it's usually understood that a cluster has shared - memory nodes.
- grid provides the infrastructure to turn large network of geographically distributed computers into a unified distributed - memory system.
- Such systems are usually heterogeneous, meaning nodes may be built from different types of hardware.

* Interconnection Networks

- An interconnection network is a communicati system that connects processors (cores) and memory in a parallel computer.
- Even if processors are very fast, a slow interconnect can make the whole system slow.
- 2 main types:
  - Shared-memory system
  - Distributed-memory system

Why interconnection is imp:
- All processors share the same memory
- many processors may try to access memory at the same time.
- if the interconnect is slow or congested, processors must wait, reducing performan

(1) Shared memory System

(1) Bus-Based interconnect

- A bus is a set of shared communication wires
- All processors and memory modules use the same bus.

Advantages: low cost, easy to add devices, simple design.

disadvantages:
- Only one devices can use the bus at a time
- As processors increase, contention increases
- Perbormance decreases for large system. it is suitable for small system

② crossbar interconnect

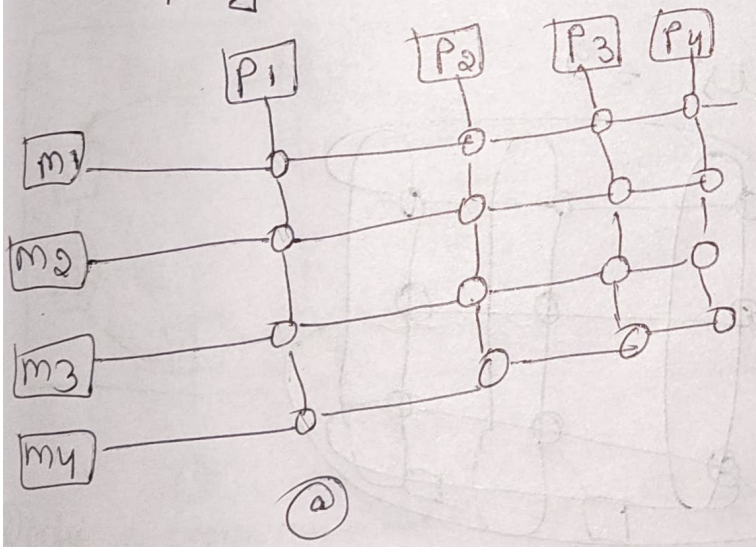
- uses switches to connect processors and memory directly
- each processor can access a different memory module simultaneously.

Advan : very fast, multiple connections can occur at the same time, less waiting

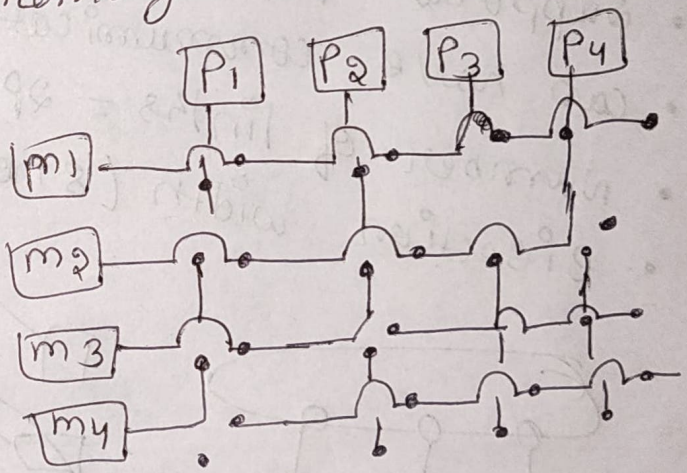
disad : expensive, complex hardware, cost increases rapidly with system size

Ex : if P1 writes to M4
P2 reads from M3
P3 reads from M1
P4 writes to M2.

All operations can happen at the same time, as long as they access different memory modules.



(a)



(b)

② Distributed - memory interconnects

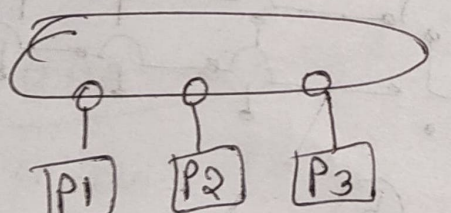
- used in distributed - memory system to handle communication b/w processor - memory pairs
- 2 main types :- • Direct interconnects
  • indirect interconnects

① Direct interconnects

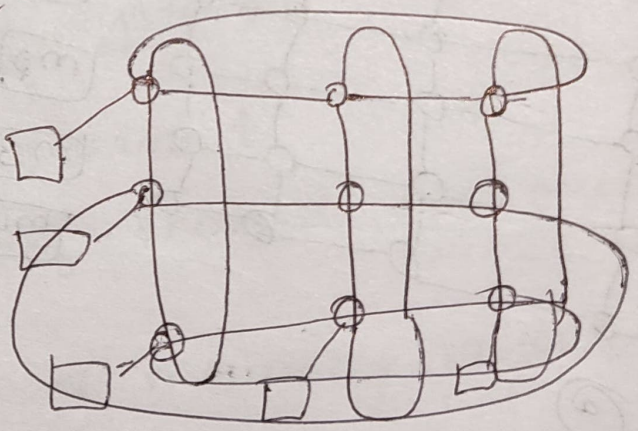
- each processor - memory pair is directly connected to a switch
- switch are also connected to other switches
  - Example : Ring, Toroidal mesh, Hypercube.

④ Ring Interconnect

- each node is connected to 2 neighbors
- supports multiple simultaneous communications
- can have communication bottlenecks
- Number of links = 2p
- Bisection width (8 nodes) = 2



ⓐ



ⓑ

ⓐ ring ⓑ toroidal mesh

⑤ Toroidal mesh interconnect

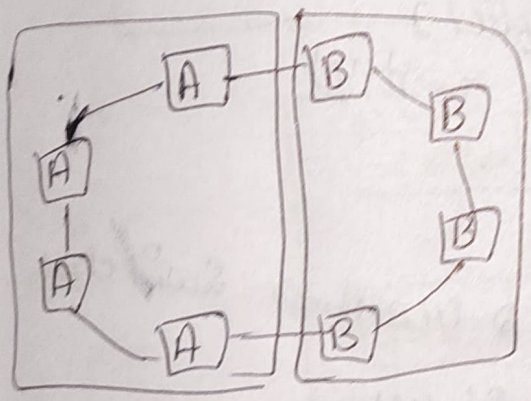
- more complex than ring
- each switch connects to 5 links

- Allows more parallel communication
- more expensive
- more number of links = $2p$
- for $p = q^2$ (q even):
  - Bisection width = $2\sqrt{p}$

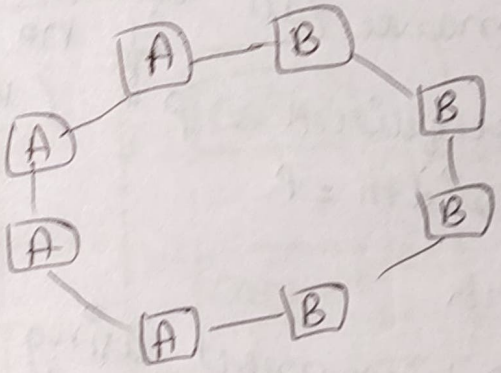
* Bisection width.
- minimum number of links removed to divide network into two equal halves
- measures communication capability. Ex:- Ring (2 links) = 2
  square toroidal mesh = $2\sqrt{p}$

* Bisection Bandwidth.
- total bandwidth of links b/w 2 halves
- shows data transfer capacity

Ex:- Ring with link bandwidth = 1 Gbps, Bisection width = 2
Bisection bw = 2 Gbps



(a)



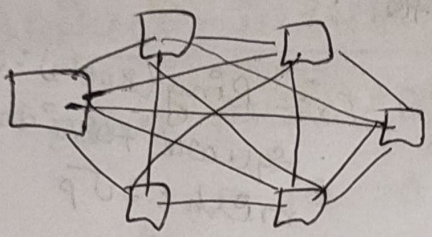
(b)

(a) only 2 communication can take place b/w the halves.

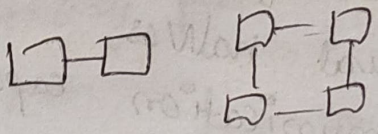
(b) four simultaneous connections can take place

(c) fully connected network
- each switch is connected to every other switch
- maximum connectivity. (bisection width = $p^2/4$)
- Impractical for large systems (requires $p^2/2 - p/2$ links)

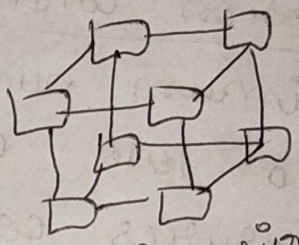
* Hypercube
- Built recursively (dimension d - 2^d nods)
- each switch connected to 1 processor + d switch
- Bisection width = P/2 (better than ring/mesh)
- Expensive : requires $\log_2(P)$ links per node



fully connected network



(a) one. (b) 2



(c) 3 - dimensional hypercubes

② Indirect interconnects

① crossbar Network
- every processor connected to every memory modu through switches
- High performance (all can communicate simultaneously if no conflict)
- Switches required = P^2 (very costly)
- Bisection width = P

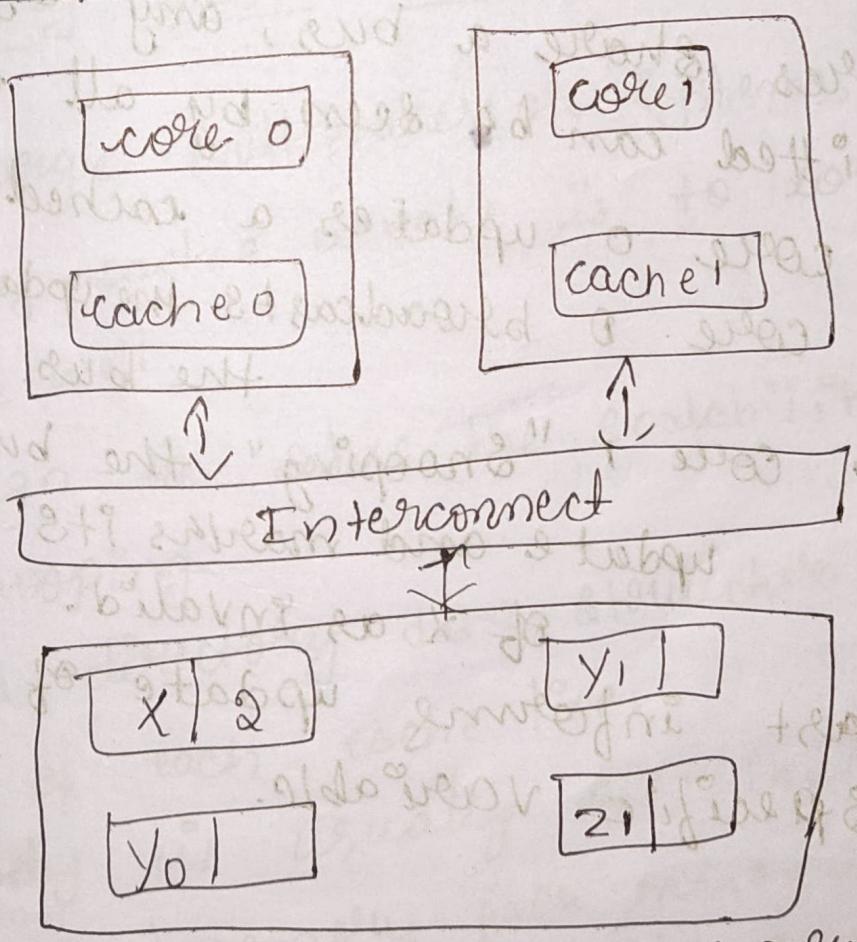
② Omega network
- multistage interconnect using 2 x 2 crossbar switches
- cheaper : requires $2P \log_2(P)$ switches
- limitations : some communications cannot occu simultaneously
- Bisection width = P/2

Ref. textbook for fig

* cache coherence

- In shared memory system, CPU caches are H/w managed, and programmers do not have direct control over cache updates.
- This lack of control leads to important issues in shared-memory systems, especially when multiple cores access and modify shared data.

Time	core 0	core 1
0	$y_0 = x;$	$y_1 = 3 * x;$
1	$x = 7;$	Statement (s) not involving x
2	Statement (s) not involving x	$z_1 = 4 * x;$



A shared-memory system with 2 cores & 2 caches

- A system with two cores, each having a private data cache.
- variables: X : shared variable
  X_0 : private to core 0
  Y_1, Z_1 : private to core 1
- when both cores read shared data, no problem occurs.
- problem arises when core 0 writes to X and then core 1 uses X in a computation

* Snooping cache coherence

Two approaches : Snooping each

- Idea is originates from bus-based syst
- When cores share a bus, any signal transmitted can be seen by all cores.

Examples :
- core 0 updates a cached value of X
- core 0 broadcasts the update across the bus.
- core 1 "snooping" the bus sees the update and marks its cached copy of x as invalid.

- Broadcast informs update of cache line not specific variable.

- Interconnect need not be bus, but must allow broadcasts.
- works with both write through & write back caches.

write through: no extra traffic (updates visible directly)

write back: extra communication needed.

- Not limited to buses, but requires an interconnect that supports broadcasts.
- simple and effective for small systems.
- limitations: Not scalable to large system because every update requires a broadcasts

* Directory based coherence

- In large multiprocessor systems, broadcasting every cache update is to costly and doesn't scale well.
- Design to solve the scalability problems of snooping
- uses a directory data structure to store the status of each cache line.
- directory is usually distributed.
- each core/memory pair manages directory entries for its own memory

* When a line is cached - directly entry
- ed with which core has it.
* When a variable is updated -> directory consulted is checked
* only the cores that have that cache line
told to invalidate their copies

* Advantage : . no need for System-wide broad
. only relevant cores are connect

* dis : requires extra storage for the director

* False Sharing

- False sharing occurs when CPU caches operat on cache lines and multiple cores update variables that lie on the same cache line.
- Even if they are logically independent, the cache System treats them as if they are shared.

* This causes unnessary invalidations & mem traffic, which hurts performances.

* Shared-memory and distributed memory (coordinating the processes/threads)

programmers have found that in shared memory coordinating the work of processes through shared data structures is easy than sending messages. In shared systems cost of scaling the network by buses is difficult & expensive.

Distributed systems such as hyper cube, & toroidal mesh are relatively inexpensive.

These are better suited for problems requiring vast amounts of data (or) computation.

* Coordinating the processes/threads

Suppose we want to add two arrays.

ie. double x[n], y[n];

```
for (int i=0; i<n; i++)
    x[i] += y[i];
```

The programmer needs to

① Divide the work among processes/thread that,

② each process/thread gets roughly the same amount of work &

③ the ~~con~~ amount of communication required is minimized.

The task a) is called as load balancing.

The process of converting a serial program into a parallel program is called parallelization.

(1) programmers also need to
(2) Arrange for the processes/threads to Synchronize
(3) Arrange for the communication among processes/threads.

* Shared memory:

In g. Shared memory programs, variables can be shared (or) private.
- Shared variables can be accessed by only one thread. the communication among threads is done through shared variables.

* Dynamic & static threads

(1) Dynamic threads: Shared-memory programs use dynamic threads.
- there is a master thread & a collection of worker threads.
- The master thread waits for work requests
- when master receives a new work request. it forks (assigns) the work to a work & joins the collection of worker threads, Here resources are efficiently utilized.

(2) static thread: Dynamic memory program mostly use static threads
- Here all the workers thread are forked by master thread whenever work arrives

- All threads run until the work is completed & gets terminated. If threads are idle, the master thread do some cleanup (free memory) & then it also terminates.
- If resources are available then static threads are more better performance than dynamic threads

* Non determinism:

A computation is non deterministic if a given input can result in different o/p. multiple threads which are executing independently execute at different rates

Two threads with ID (or rank) 0 & with ID (or rank) 1 have private variable my-x.

Thread 0's value for my-x is 7.
Thread 1's value for my-x is 19
Suppose both threads executes the following code
printf ("Thread %d > my-x = %d \n", myrank, my-x);

output: Thread 0 > my-x = 7
Thread 1 > my-x = 19

but it could be also be
Thread 1 > my-x = 19
Thread 0 > my-x = 7.

- The time for execution of statements vary for one thread to other threads.
- The order of execution can be predicted.

Ex: If both threads want to add the values stored in ~~memory~~ my_val into shared-memory location x. Both threads execute.

```
my_val = compute_val (my_rank);
x += my_val.
```

Assum values are loaded from main memory directly into register & vice/versa.

Possible sequence of events:

Time	core 0	core 1
0	Finish assignment to my_val	In call to compute_val
1	Load x=0 into register	Finish argument to my_val
2	load my_val = 7	load x=0 into register
3	Add my_val = 7 to x	load my_val = 19
4	Store x = 7	Add my_val to x.
5	start other work	Store x = 19.

Note: gives incorrect value.

Race Condition: When threads or processes attempt to simultaneously access a shared resource & accesses result is an error, this condition is called as race condition.
We need each thread's operation to be atomic. Atomic means thread has completed the operation & no other thread has modified the memory location.

Critical Section: A block of code that can only be executed by one thread at a time is called a critical section.

Mutual Exclusion: If one thread is executing in its critical section, then other threads are excluded from execution. The mutual exclusion is done by mutual exclusion locks or mutex or locks. The critical section is protected by locks.

* Mutex enforces serialization of the critical section. Since one thread executes at a time in critical section, this code is effectively serial. Critical sections must be few & short.

* Busy-waiting: These are alternatives to mutex. Thread enters a loop to test a condition.

```
my_val = compute_val(my_rank)
if (my_rank == 1)
   while (!ok_for_1); // busy_wait loop
   x += my_val ;      // critical section
if (my_rank == 0)
      ok_for_1 = true; // let thread 1 update x
```

Until thread 0 executes ok_for_1 = true, thread 1 will be stuck up in the loop while(!ok_for_1). This loop is called as busy-wait.

* Thread Safety:

The C string library function strtok() splits an input string into substrings. Ex if thread 0 calls strtok function & then thread 1 calls strtok before thread 0 has completed splitting its string. Then thread 0's string will be lost or overwritten & thread 1's string will be appeared.

The function strtok() is not thread safe. Hence serial programs can be used in multi-threaded programs and are considered thread safe.

* Distributed Memory:

In distributed-memory programs, the cores can directly access their own private memories. Several API's are used here like message-passing.

* Message-passing:

A message-passing API provides a send and a receive function. process 1 can send a message to process 0 with the code.

```
char message[100];
my_rank = get_rank();
if (my_rank == 1)
{ sprintf(message, "Greeting from process 1");
  Send(message, msg_char, 100, 0);
} else if (my_rank == 0)
{ receive(message, msg_char, 100, 1)
} printf("process0> Received: %s\n", message);
```

Here get_rank function returns the calling process's rank.

* One-sided Communication: (Remote memory access)

A single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process.

→ Reduces the cost of communication
→ " " overhead

* Partitioned global address space:

→ Mechanisms for shared memory programs
→ private variables are allocated in local memory.
→ Determine array elements of the respective process.

module - 1

* Vector processors

Vector processors operate on arrays (or) vectors of data while conventional CPU's operate on individual data elements (or) scalars.

Typical recent systems have the following characteristics

① Vector registers: Store a vector of operands & operate simultaneously on their contents. Vector length range from 4 to 256 64 bit elements

② Vectorized & pipelined functional units: Same operation is applied to each pair of the elements in the vector, thus vector operations are SIMD

③ Vector instructions: These operate on vectors of vector_length on loops such as

```
for (i=0; i<n; i++)
    x[i] += y[i];
```

require only a single load, add & store for each block of vector_length elements

④ Interleaved memory: The memory system consists of multiple banks that can be accessed independently. there is little to no delay in loading/storing successive elements.

⑤ Strided memory access & H/w Scatter/gath

In strided memory access, the pgm accesses elements of a vector located at fixed intervals for ex: accessing the 1st element, 5th element & ninth element, & so on would Scattered access with a stride of four.

- Scatter/gather is writing (scatter) or reading (gather) elements of a vector located at irregula intervals.

for example, accessing 1st element, 2nd element, 4th element, 8th element & so, on

- typically vector systems provide special hardwar to accelerate strided access & scatter/gathe

→ vector processors are very fast & easy to use

→ vectorized compilers are good at identifying code that can be vectorized.

→ Identify loops that cannot be vectorized

→ Vector system have very high memory bandwidth

→ vector system cant handle irregular data structures.

* Graphics processing units (GPU):

Real-time API's use points, lines & triangles to internally represent the surface of an object. they use a graphics processing pipline to convert the internal representation into an array of pixels that can be sent to a computer screen. Several of the stages of this pipeline are program-able.

The behavior of the programable stages is specified by functions called shader functions. which are short (or) few lines of c code.

Shader functions are implicitly parallel & can be applied to multiple elements (vertices) in the graphics stream.

→ GPU's optimize performance by using SIMD parallelism & do which is obtained by using large number of datapaths (128), on each GPU core

→ As GPU require large MB of data to process image, hence need to maintain high rates of data movement to avoid states (or) delays.

→ GPU rely on hardware multithreading which store the state of hundreds of suspended threads

→ Drawback of GPU is many threads processing lots of data need to keep datapaths busy hence GPU's give poor performance on small problem

→ GPU's are not pure SIMD nor MIMD
→ GPU's can use shared memory or, distributed memory
→ GPU's are used in high performance computing

---

* Latency & Bandwidth

The latency is the time that elapses b/w the Source's transmitting it's first byte data & destinations receiving that first byte data.

If latency of interconnect is l seconds & the bandwidth is 'b' bytes per seconds, then the time it takes to transmit a message of n bytes is

message transmission time = $l + n/b$