# Unit-I

D.Venkata Vara Prasad

## MCA…

- Why we need ever-increasing performance!
- Why we’re building parallel systems
- Why we need to write parallel programs.

## Why we need ever-increasing performance

- Computational power is increasing, but so are our computation problems and needs.
- Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.
- More complex problems are still waiting to be solved.

## Changing times

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.
- Since then, it’s dropped to about 20% increase per year.

## Why we’re building parallel systems

- Up to now, performance increases have been attributable to increasing density of transistors.
- But there are inherent problems.

## Limitations of Single core …

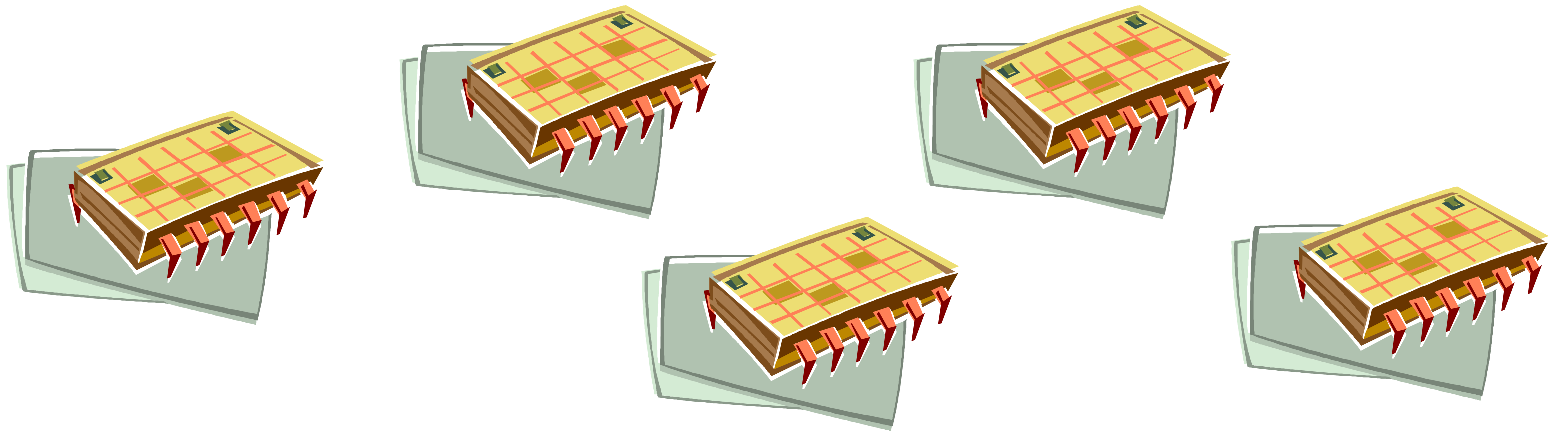
- Smaller transistors = faster processors.
- Faster processors = increased power consumption.
- Increased power consumption = increased heat.
- Increased heat = unreliable processors.

## Limitations of Single core…

- A simple Thump rule is that
  - For every 1% rise in the clock frequency you will see 3% rise in the power consumption
  - Thus the heat dissipation also increases.

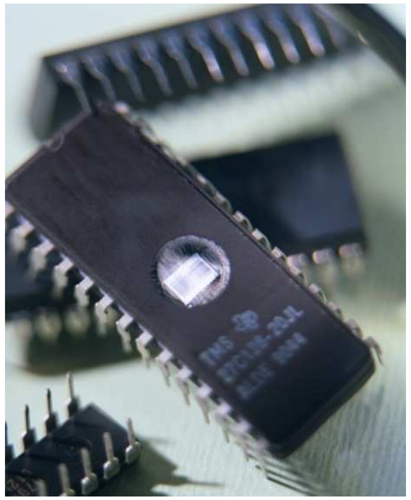
## An intelligent solution…

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.



## An intelligent solution

- Move away from single-core systems to multicore processors.
- “core” = central processing unit (CPU)
- Introducing parallelism!!!



## Why Multicores ?

- Difficult to make single core clock frequency higher
- Many new applications are multithreaded
- General trend in Computer Architecture  is shift toward more parallelism

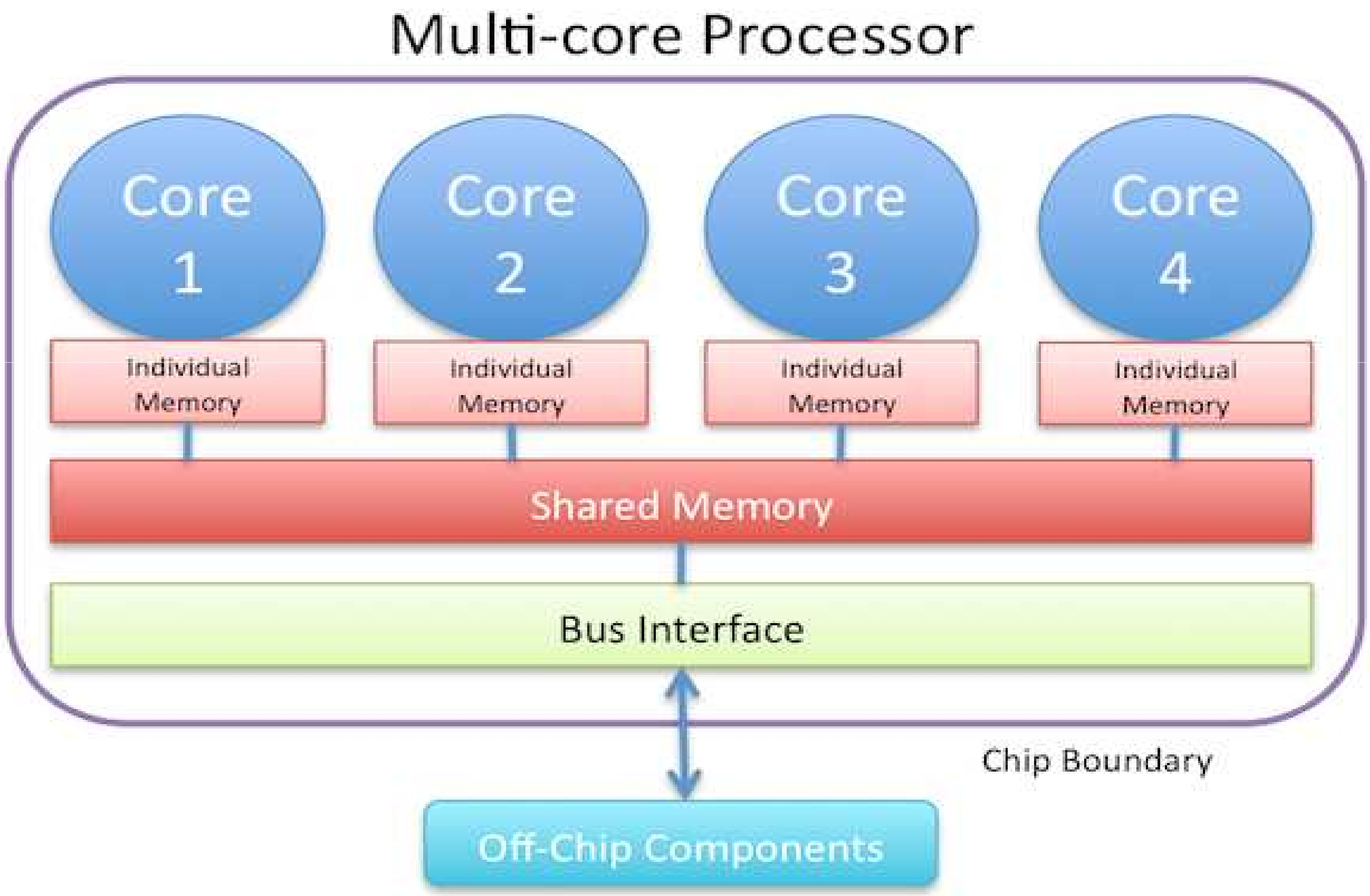
## Multicore Architectures …

- Multi-core is a design in which a single physical processor contains the core logic of more than one processor.
- It’s a special kind of Multiprocessor.
- All processors are on the same chip

## Multicore Architectures…

- Multicore processors are MIMD
- Different cores executes different threads ( Multiple Instructions) ,operates on different parts of memory( Multiple Data)
- Multicore is a Shared Memory Multiprocessors. All cores share the same memory.

# Multicore Architectures…



## Multicore Architectures…

- contain two or more distinct cores in the same physical package
- each core has its own execution pipeline
- each core has the resources required to run without blocking resources needed by the other software threads.
- core design enables two or more cores to run at somewhat slower speeds and at much lower temperatures

## Multicore Architectures

- combined throughput of these cores delivers processing power greater than the maximum available today on single-core processors and at a much lower level of power consumption
- Ex: 16 core MIT RAW processor operates at 425 MHz can perform 100 time the number of operations per second than Intel Pentium-3 with 600MHz.

# Advantages

- Occupies less space on PCB
- Higher throughput
- Consume less power
- Cache coherency can be greatly improved
- Performs more operations/sec with less frequency

## Disadvantages

- Maximizing the utilization of the computing resources provided by multi-core processors requires adjustments both to the operating system (OS) support and to existing application software

- They are more difficult to manage thermally than lower-density single-chip designs

## Multicore applications

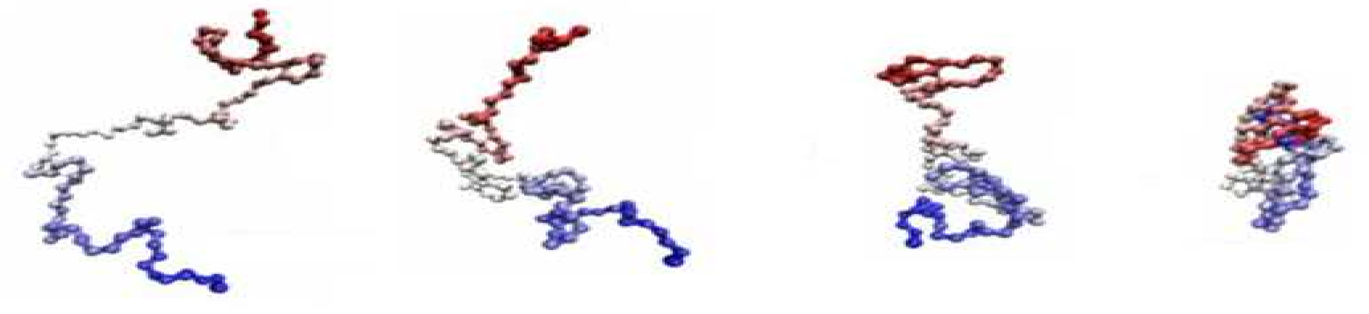
- Data base servers
- Web servers
- Compilers
- Multimedia Applications
- Scientific Applications
- General applications with TLP as opposed to ILP
- Downloading s/w while running Anti virus s/w
- Editing photo while recording TV show.

## Climate modeling

- To understand climate change:
  - we need far more accurate computer models
  - models that include interactions between the atmosphere, the oceans, solid land, and the ice caps at the poles.

# Protein folding

- To analyze the protein structures:
  - ability to study configurations of complex molecules such as protein
  - misfolded proteins may be involved in diseases such as Parkinson’s, and Alzheimer’s etc.



## Drug discovery

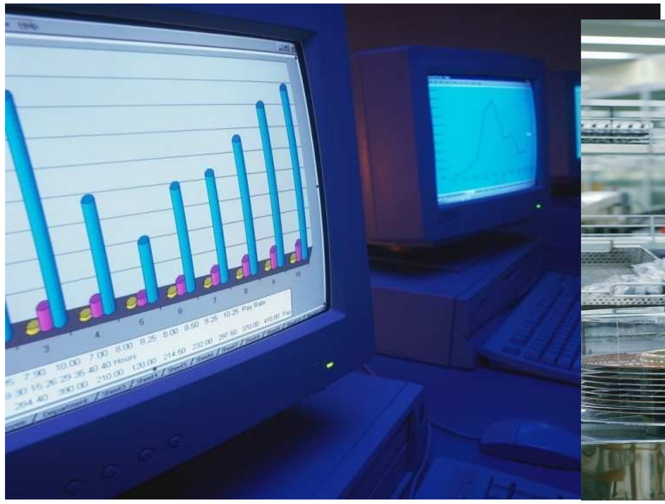
- increased computational power can be used in research into new medical treatments.
- devise alternative treatments by careful analysis of the genomes of the individuals for whom the known treatment is ineffective.

## Energy Research

- •Increased computational power will make it possible to program much more detailed models of technologies such as wind turbines, solar cells, and batteries.
- •may provide the information needed to construct far more efficient clean energy sources

## Data analysis

- •The quantity of data stored worldwide doubles every two years.
- •The vast majority of it is largely useless unless it’s analyzed
- •Ex: knowing the sequence of nucleotides in human DNA is, by itself, of little use.
- •  Understanding how this sequence affects development and how it can cause disease requires extensive analysis.



# Why we need to write parallel programs

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to run faster.

## Approaches to the serial problem

- Rewrite serial programs so that they're parallel.

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

## Example

- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

## Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

**Each core uses it's own private variables & executes this block of code independently of the other cores.**

# Example (cont.)

- After each core completes execution of the code, is a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.

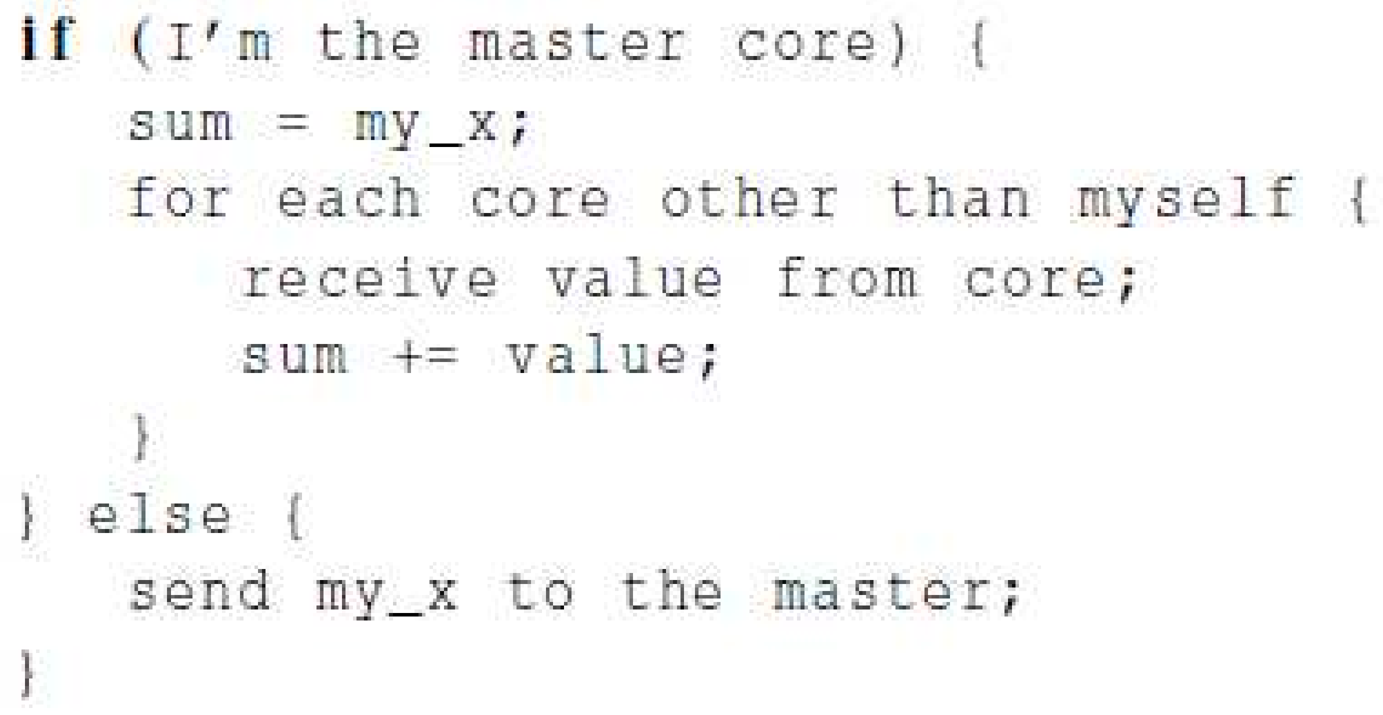
- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

1,4,3,   9,2,8,   5,1,1,   5,2,7,   2,5,0,   4,1,8,   6,5,1,   2,3,9

## Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

# Example (cont.)



```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

## Example (cont.)

| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 95 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

**But wait!**

**There’s a much better way to compute the global sum.**

## Better parallel algorithm

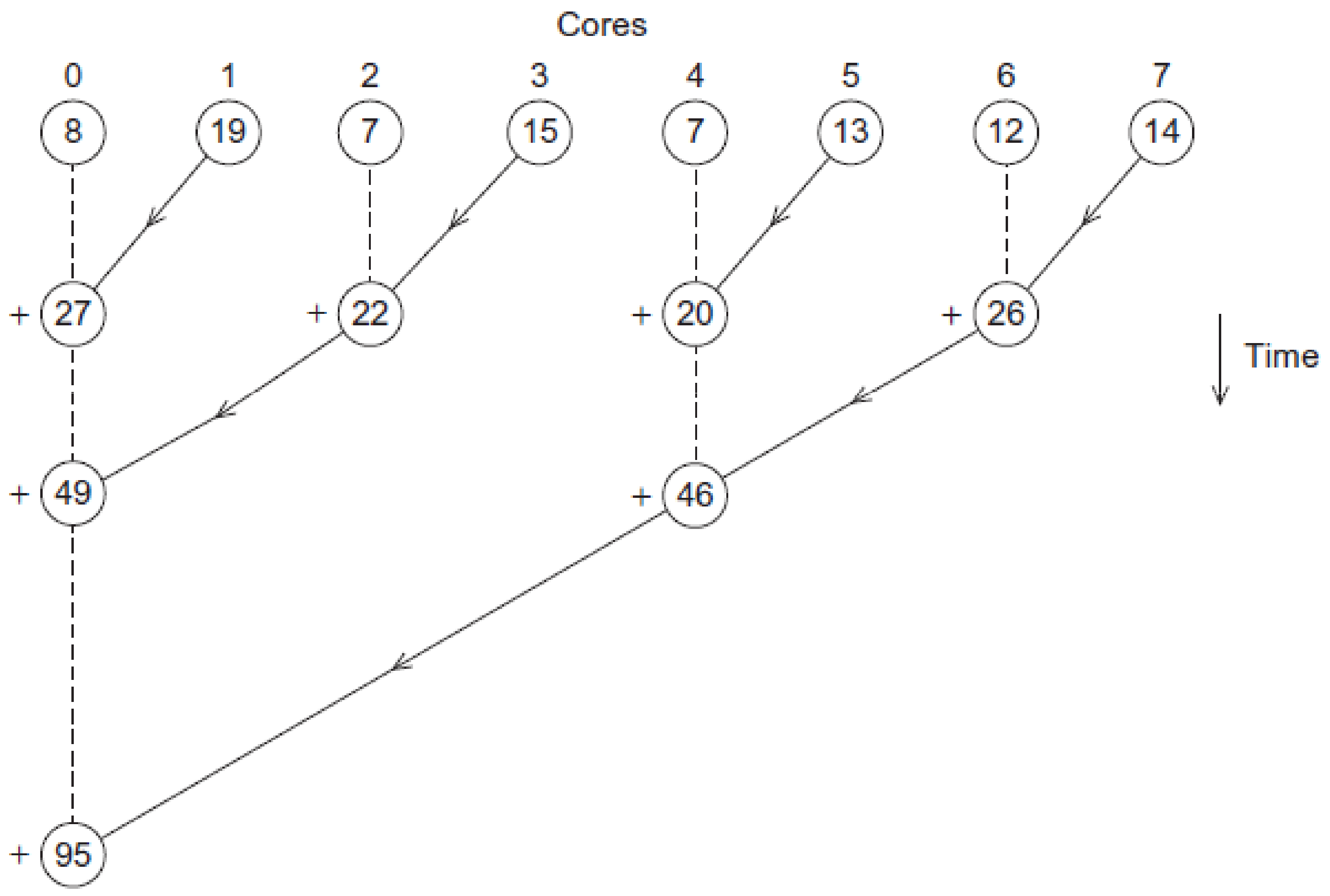
- Don’t make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1’s result.
- Core 2 adds its result with core 3’s result, etc.
- Work with odd and even numbered pairs of cores.

## Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

# Multiple cores forming a global sum



## Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2!

## Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.

- That’s an improvement of almost a factor of 100!

## How do we write parallel programs?

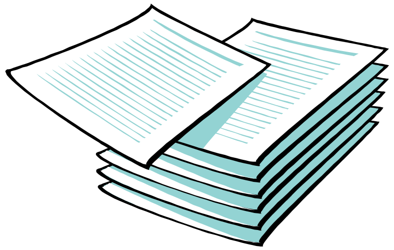
- Task parallelism
  - Partition various tasks carried out solving the problem among the cores.

- Data parallelism
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it’s part of the data.

# Professor P

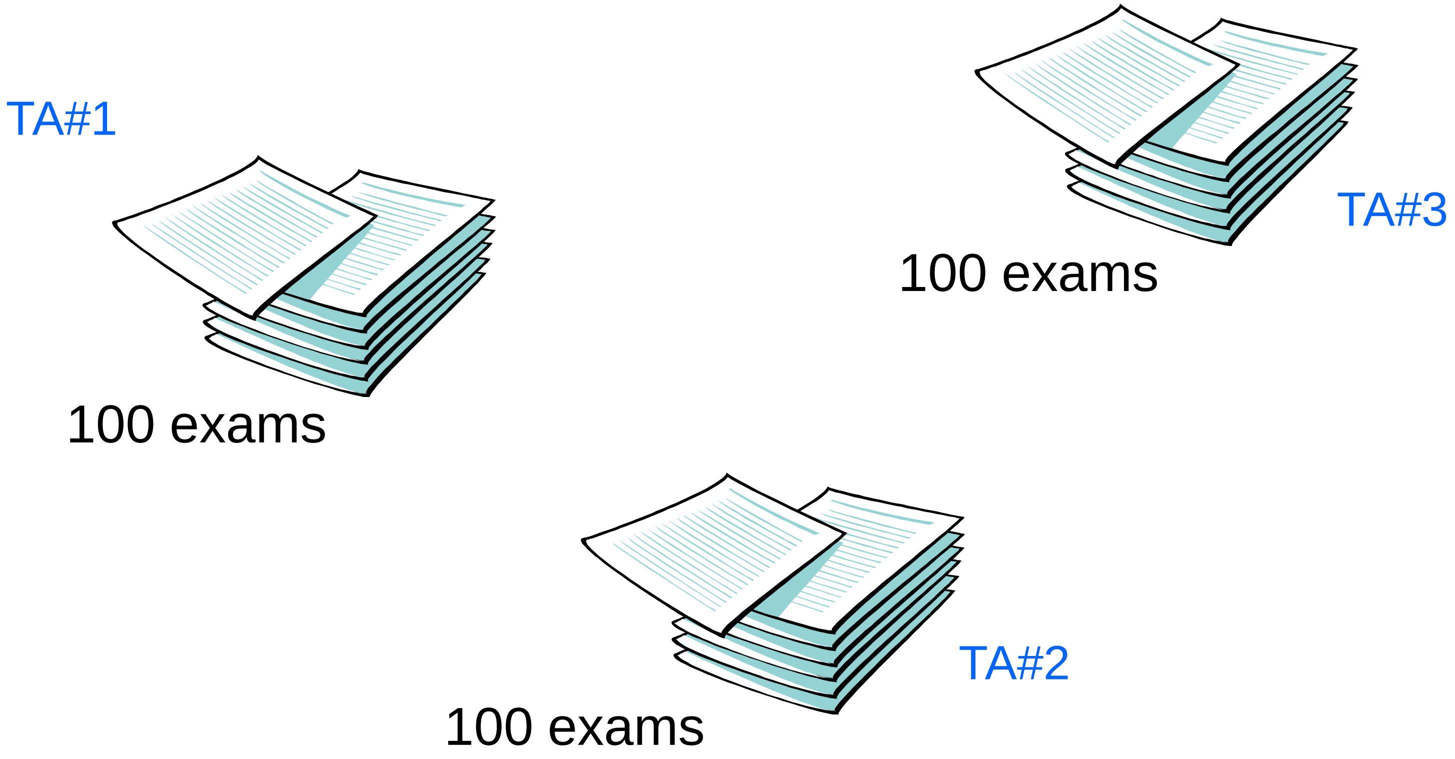
15 questions

300 exams

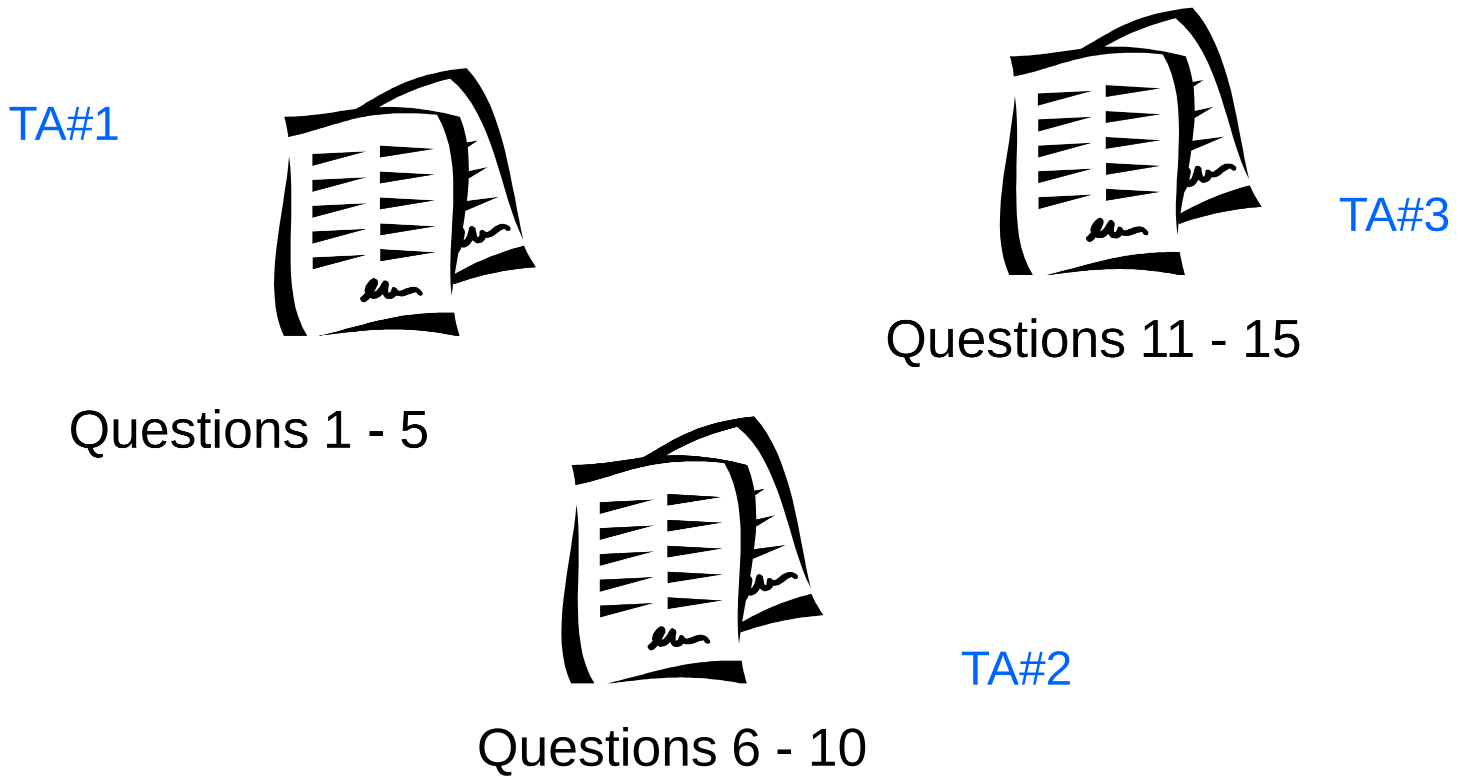


## Professor P’s grading assistants

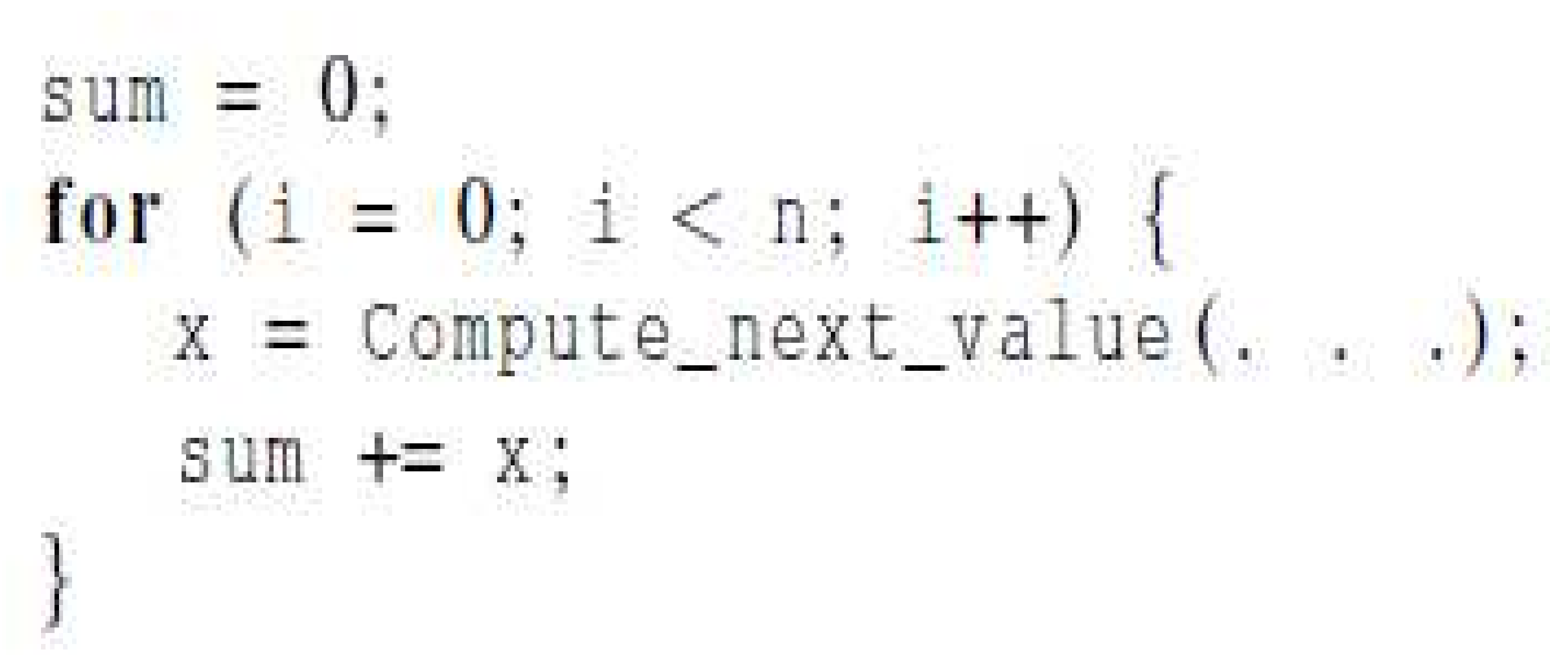
# Division of work – data parallelism



# Division of work – task parallelism

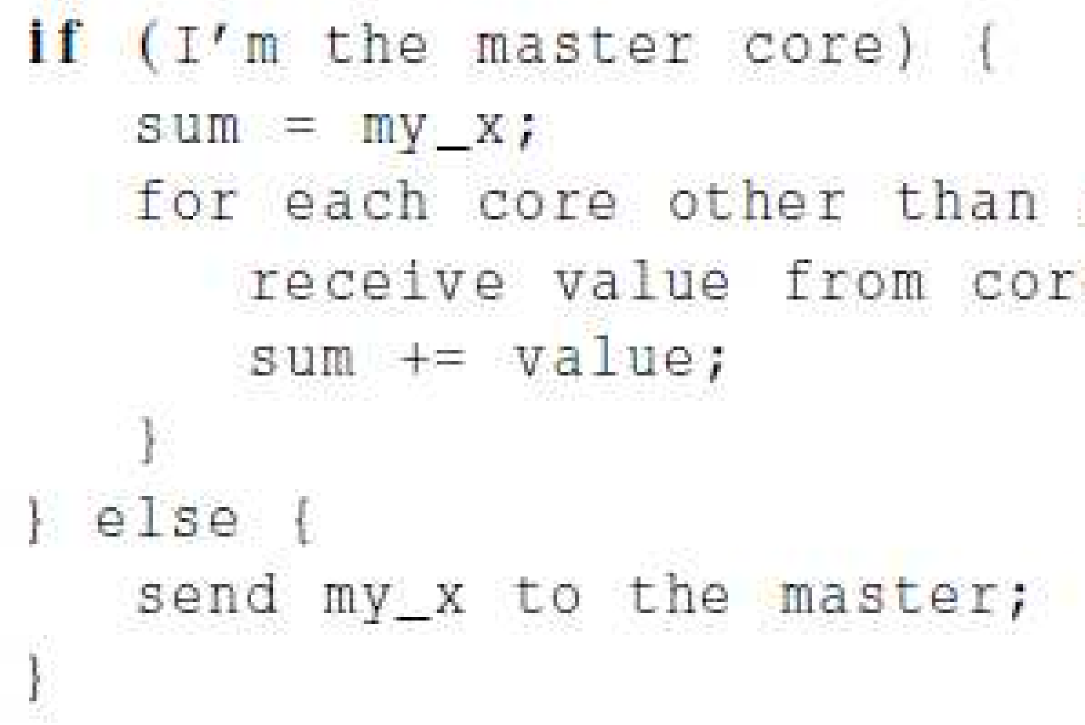


# Division of work – data parallelism



```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

# Division of work – task parallelism



```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

<u>Tasks</u>

1) Receiving
2) Addition

## Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

## What we’ll be doing

- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using three different extensions to C.
  – Message-Passing Interface (MPI)
  – Posix Threads (Pthreads)
  – OpenMP

## Flynn’s Classification

classic von Neumann

| | |
|---|---|
| SISD<br>Single instruction stream<br>Single data stream | (SIMD)<br>Single instruction stream<br>Multiple data stream |
| MISD<br>Multiple instruction stream<br>Single data stream | (MIMD)<br>Multiple instruction stream<br>Multiple data stream |

not covered

## Flynn’s Classification

- Single instruction stream, single data stream (SISD)
  - von Neumann
- Single instruction stream, multiple data streams (SIMD)
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units

## Flynn’s Classification

- Multiple instruction streams, single data stream (MISD)
  - No commercial implementation

- Multiple instruction streams, multiple data streams (MIMD)
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

## SIMD

- SIMD Arch have significant DLP
- Single Instruction can launch many data opns
- SIMD is more energy efficient than MIMD
  - MIMD needs to fetch and execute one instruction per data opns.
- SIMD is more attractive for PMDs.
- Advantage of SIMD over MIMD
  - Programmer thinks sequential execution yet achieves parallel speedup by having parallel data operations

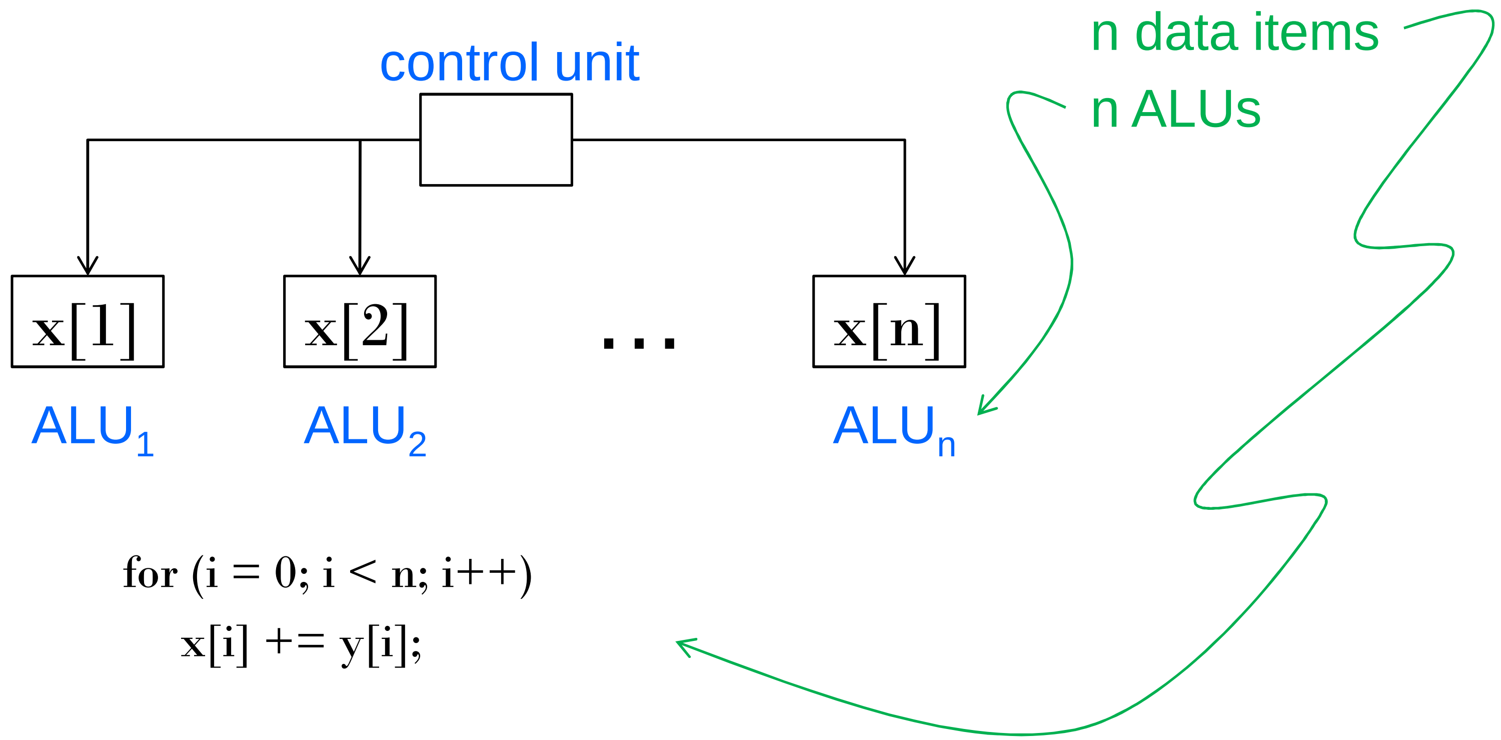
## SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.

## SIMD

- SIMD has 3 variations:
  - Vector Architectures
    - Allows pipelined execution of many data operations
  - SIMD MMX
    - Allows simultaneous parallel data operations that support  Multimedia applications.
  - GPU Architectures
    - They offer higher performance than traditional multicore
    - They have system processor, system memory & graphics memory.

## SIMD example



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

## SIMD

- What if we don’t have as many ALUs as data items?
- Divide the work and process iteratively.
- Ex. m = 4 ALUs   and   n = 15 data items.

| Round3 | $ALU_1$ | $ALU_2$ | $ALU_3$ | $ALU_4$ |
|---|---|---|---|---|
| 1 | X[0] | X[1] | X[2] | X[3] |
| 2 | X[4] | X[5] | X[6] | X[7] |
| 3 | X[8] | X[9] | X[10] | X[11] |
| 4 | X[12] | X[13] | X[14] | |

## SIMD drawbacks

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

## Vector Processors

- Efficient way to execute a vectorizable application is by Vector processor- Jim Smith
- ***vector processor*** is a CPU that executes instructions that operate on arrays of data.
  - It collects set of data elements put them in the register file
  - operates on the data in those register files and stores the results back in memory.
  - These reg files acts like buffers and hide the memory latency.

## Vector processors

- SIMD classification
- Also be called as **array processor**
- Improves performance on numerical simulations
- Used in Video game console and Graphic accelerators
- Ex: VIS, MMX, SSE, AltiVec and  AVX

## Vector processors

- Operate on arrays or vectors of data while conventional CPU’s operate on individual data elements or scalars.

- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.

## Vector processors

- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector (or pairs of elements).
- Vector instructions.
  - Operate on vectors rather than scalars.

## Vector processors

- Interleaved memory.
  - Multiple “banks” of memory, which can be accessed more or less independently.
  - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather.
  - The program accesses elements of a vector located at fixed intervals.

## How Vector Processors Work: An Example

- Let's take a typical vector problem

  Y = a* X+ Y

- X and Y are vectors resident in memory
- a is a scalar
- This problem is the so-called SAXPY or DAXPY
  - SAXPY –single precision a * X plus Y
  - DAXPY -double precision a * X plus Y

## How Vector Processors Work: An Example

- MIPS code for the DAXPY loop

```
•           L.D          FO,a                  ;load scalar a
•           DADDIU    R4,Rx,#512            ;last address to load
• Loop:  L.D           F2,0(Rx)     ;load X[i]
•          MUL.D       F2,F2,FO    ;a x X[i]
•           L.D          F4,0(Ry)     ;load Y[i]
•           ADD.D      F4,F4,F2     ;a x X[i] + Y[i]
•           S.D          F4,9(Ry)     ;store into Y[i]
•           DADDIU   Rx,Rx,#8     ;increment index to X
•           DADDIU   Ry,Ry,#8     ;increment index to Y
•           DSUBU     R20,R4,Rx  ;compute bound
•           BNEZ        R20,Loop   ;check if done
```

## How Vector Processors Work: An Example

- VMIPS code for DAXPY
  - L.D FO,a ;load scalar a
  - LV V1,Rx ;load vector X
  - MULVS.D V2,V1,F0 ;vector-scalar multiply
  - LV V3,Ry ;load vector Y
  - ADDVV.D V4,V2,V3 ;add
  - SV V4,Ry ;store the result

## How Vector Processors Work: An Example

- vector processor reduces the instruction bandwidth
- executes only 6 instructions vs almost 600 for MIPS
- It occurs because the vector operations work on 64 elements
- overhead instructions that constitute half the loop on MIPS are not present in the VMIPS code
- compiler produces vector instructions for such a sequence
- resulting code spends its time running in vector mode , the code is said to be vectorized or vectorizable.

## How Vector Processors Work: An Example

- Loops can be vectorized when they do not have dependences between iterations of a loop, are called loop-carried dependences.
- Another important difference between MIPS and VMIPS is the frequency of pipeline interlocks.
- In the MIPS code, every ADD. D must wait for a MUL. D, and every S. D must wait for the ADD. D.
- On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline.

## How Vector Processors Work: An Example

- Vector architects call forwarding of element dependent operations chaining, in that the dependent operations are "chained" together.
- Thus, pipeline stalls are required only once per vector instruction, rather than once per vector element.

## Vector processors - Pros

- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
- Compilers also can provide information about code that cannot be vectorized.
  - Helps the programmer re-evaluate code.
- High memory bandwidth.
- Uses every item in a cache line.

## Vector processors - Cons

- They don’t handle irregular data structures as well as other parallel architectures.

- A very finite limit to their ability to handle ever larger problems. (scalability)

## Graphics Processing Unit

- **A graphics processing unit** (GPU), is similar CPU
- Designed specifically for performing the complex mathematical and geometric calculations that are necessary for graphics rendering.

## Graphics Processing Unit

- A graphics processing unit (GPU) is a computer chip that performs rapid mathematical calculations, primarily for the purpose of rendering images.
- occasionally called **visual processing unit** (**VPU**)
- GPU is able to render images more quickly than a CPU because of its parallel processing architecture
- Nvidia introduced the first GPU, the GeForce 256, in 1999
- Others include AMD, Intel and ARM.
- In 2012, Nvidia released a virtualized GPU, which offloads graphics processing from the server CPU in a virtual desktop infrastructure.

## Graphics Processing Unit

- GPUs are used in
  - Embedded Systems
  - Mobile phones
  - Personal computers
  - Workstations
  - Game consoles

# GPU Vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds
- A GPU is for the most part deterministic in its operation
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

## What  are GPU’s Growth?

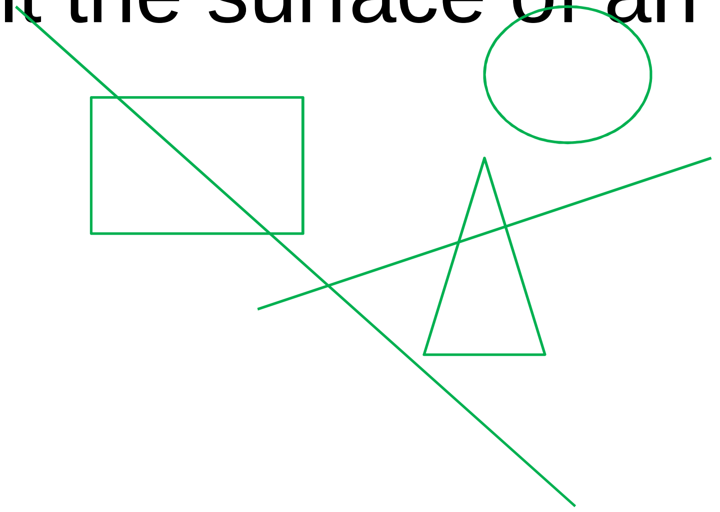
- Entertainment Industry has driven the economy of these chips?
  - Males age 15-35 buy $10B in video games / year
- Moore’s Law ++
- Simplified design (stream processing)
- Single-chip designs

## GPU

- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- Not  Efficient For
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

# Graphics Processing Units (GPU)

- Real time graphics application programming interfaces or API’s use points, lines, and triangles to internally represent the surface of an object.



## GPUs

- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.

- Several stages of this pipeline (called shader functions) are programmable.
  - Typically just a few lines of C code.

## GPUs

- Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

- GPU’s can often optimize performance by using SIMD parallelism.
- The current generation of GPU’s use SIMD parallelism.
  – Although they are not pure SIMD systems.

## MIMD

- Supports **m**ultiple **i**nstruction streams operating on **m**ultiple **d**ata streams.

- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

## Type of parallel systems

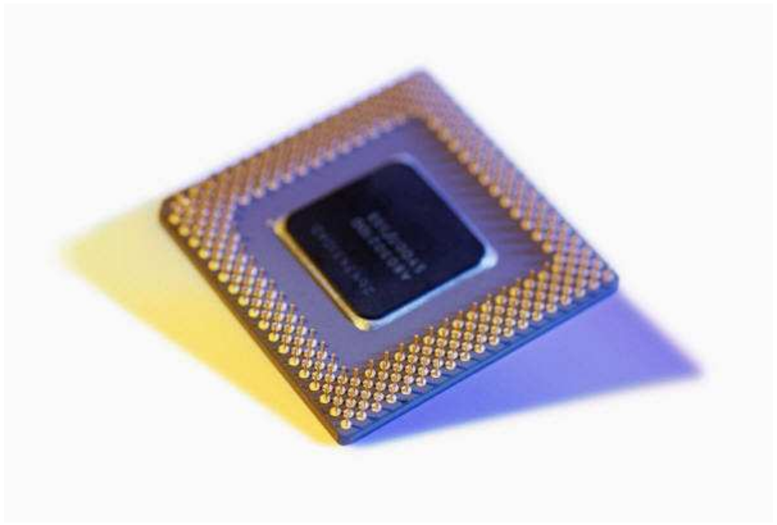
- Shared-memory
  - The cores can share access to the computer’s memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

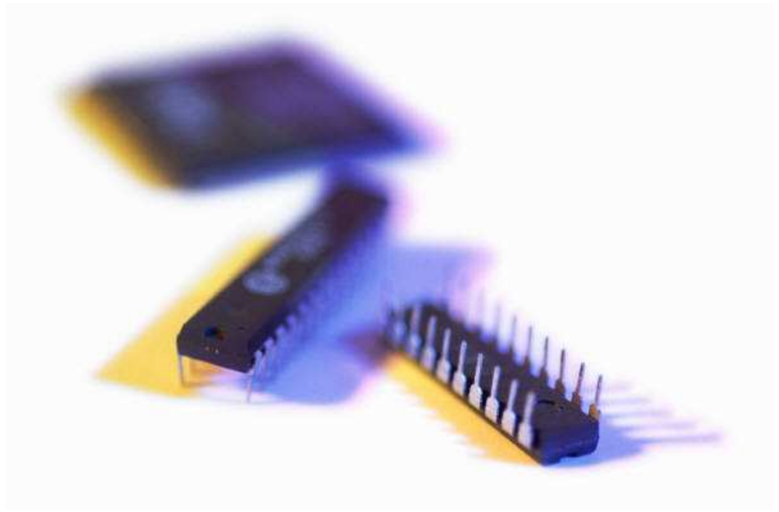
## Shared Memory System (1)

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

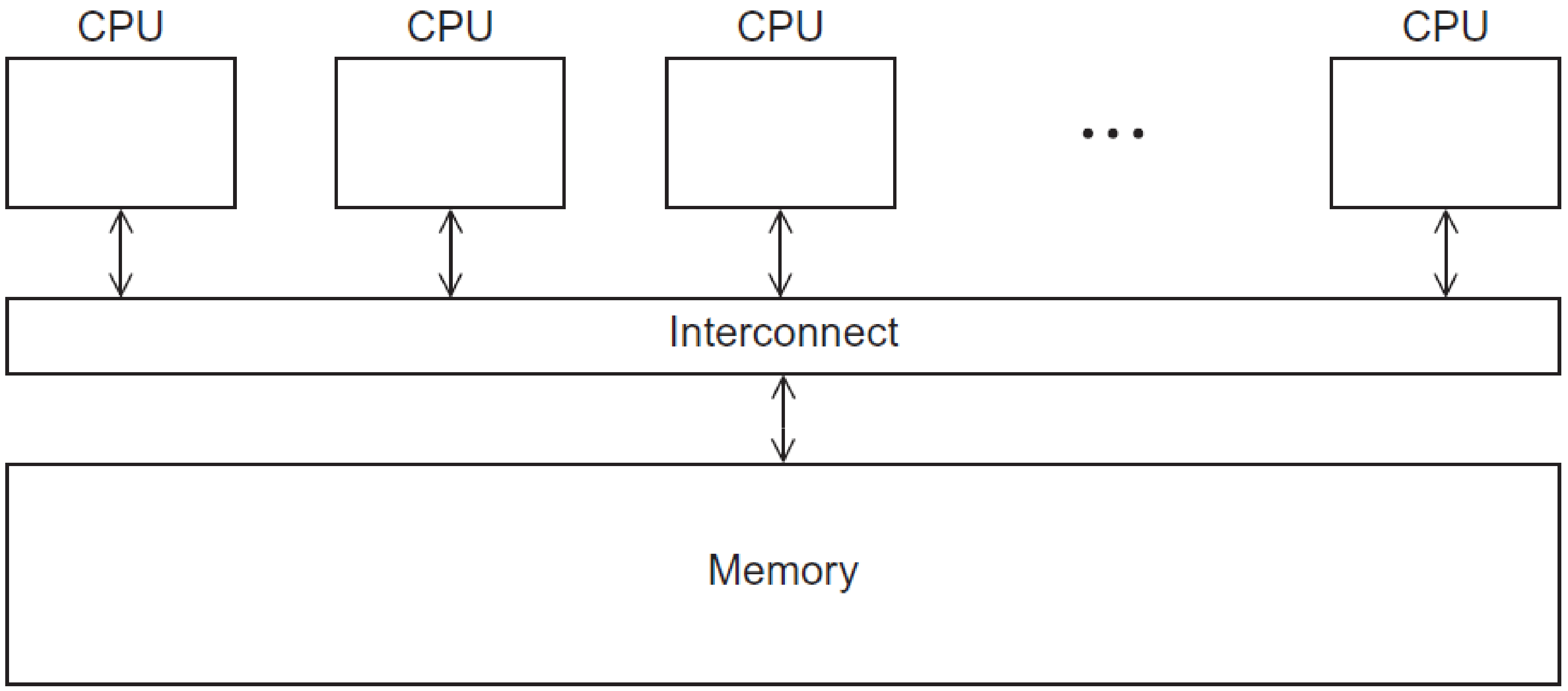
## Shared Memory System (2)

- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU's or cores on a single chip)

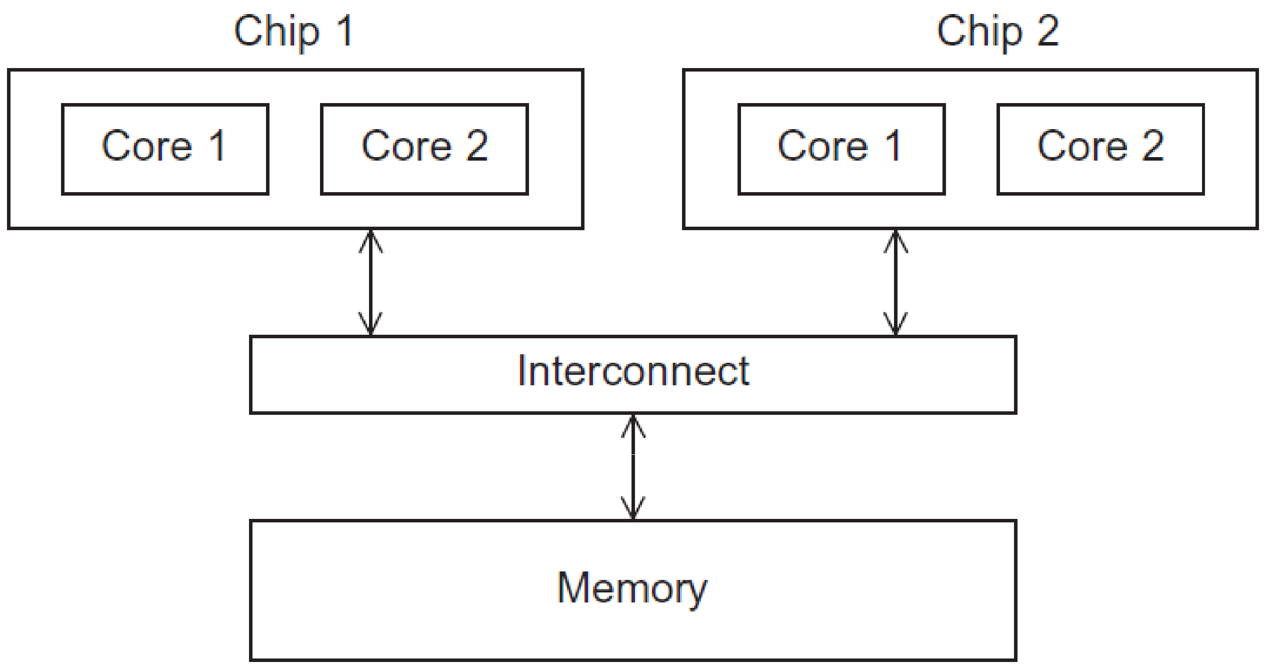




# Shared Memory System

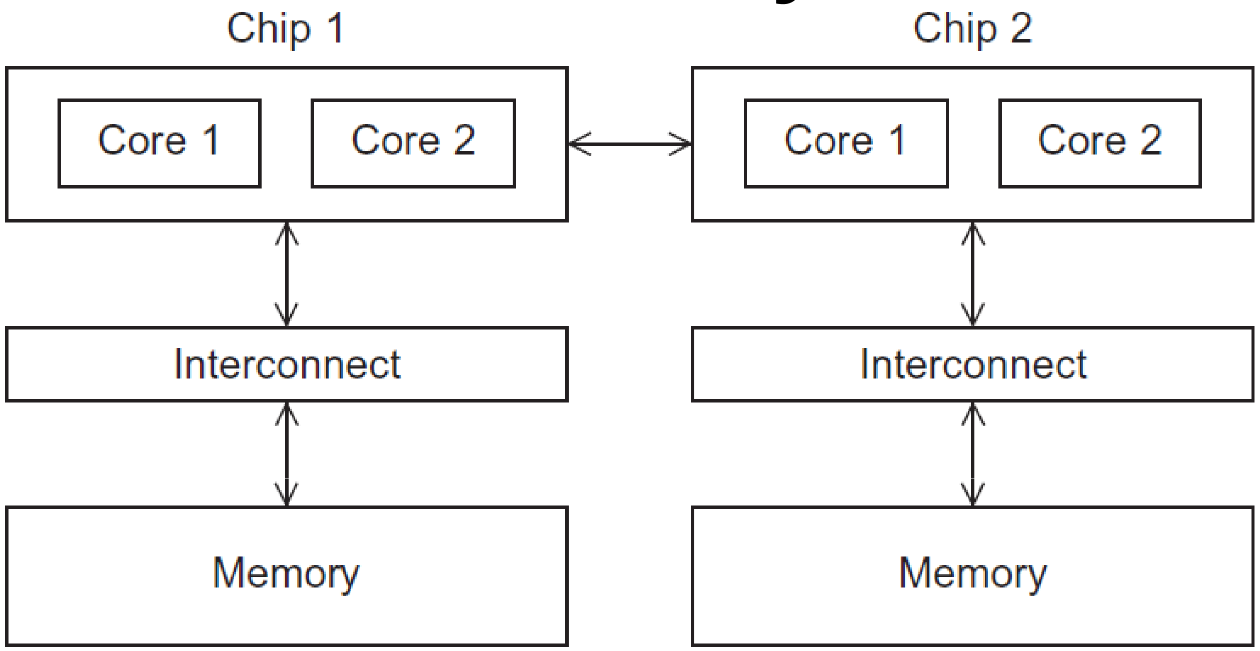


## UMA multicore system



Time to access all the memory locations
will be the same for all the cores.

## NUMA multicore system



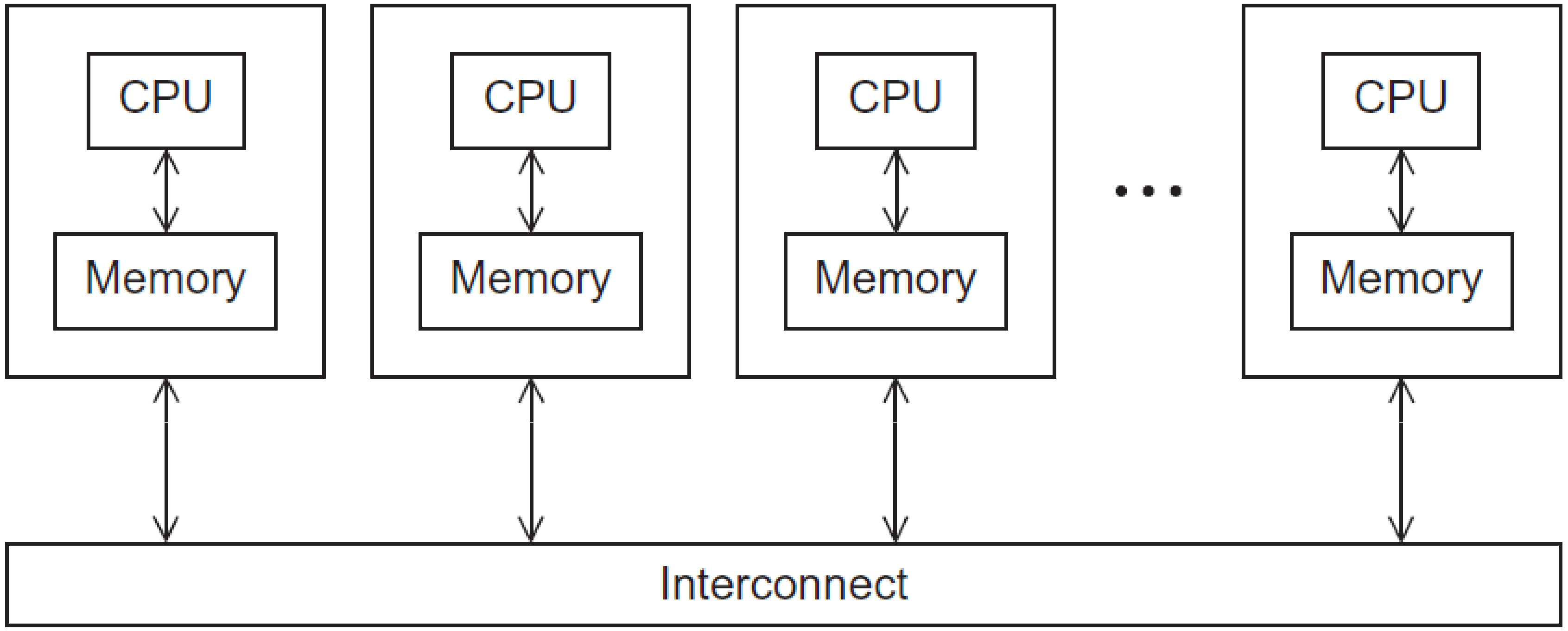
A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

## Distributed Memory System

- Clusters (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.

- Nodes of a cluster are individual computations units joined by a communication network.

# Distributed Memory System



## Interconnection networks

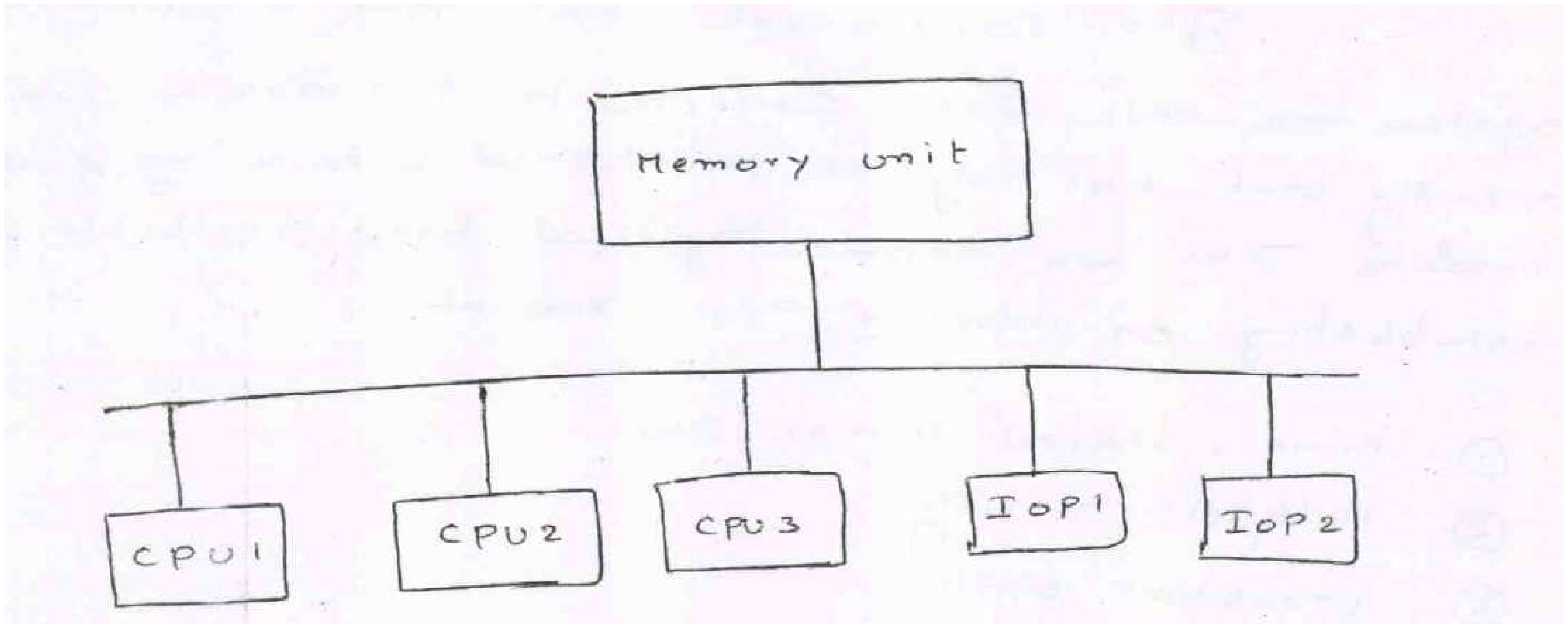
- Affects performance of both distributed and shared memory systems.

- Two categories:
  - Shared memory interconnects
  - Distributed memory interconnects

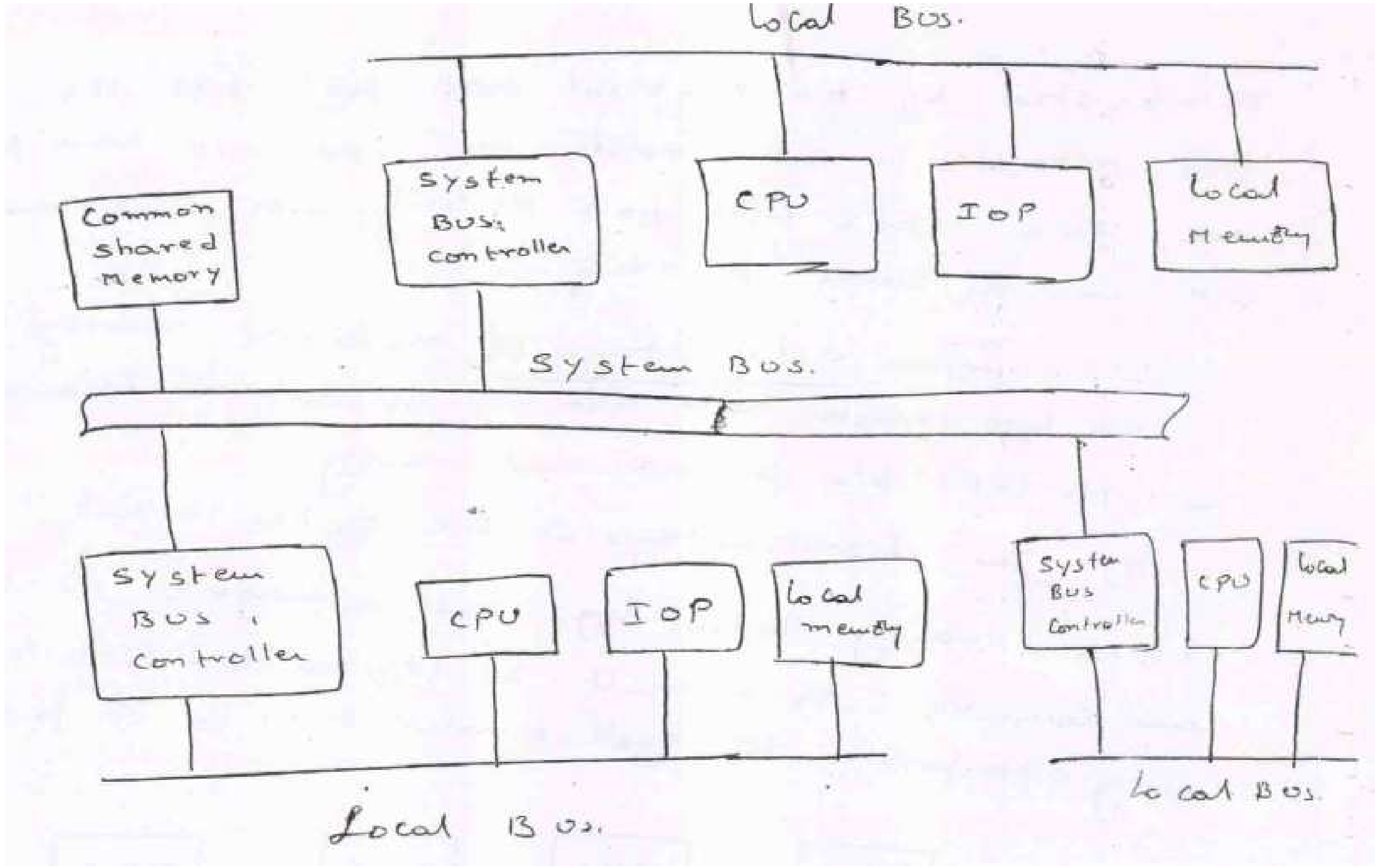
## Shared memory interconnects

- Bus interconnect
  - A collection of parallel communication wires together with some hardware that controls access to the bus.
  - Communication wires are shared by the devices that are connected to it.
  - As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

# Time shared common bus



# Dual Bus structure



## Time shared common bus

- Simple interconnection network
- Lowest cost
- Expansion of the system will degrade the performance
- Performance depends on the bandwidth of the bus
- Modifications to the system is easy
- More efficient for multiprocessor system with n=2

## Bus

•Simple +

• Cost effective for a small number of nodes Easy to implement coherence (snooping) - Not scalable to large number of nodes (limited bandwidth, electrical loading □ reduced frequency)

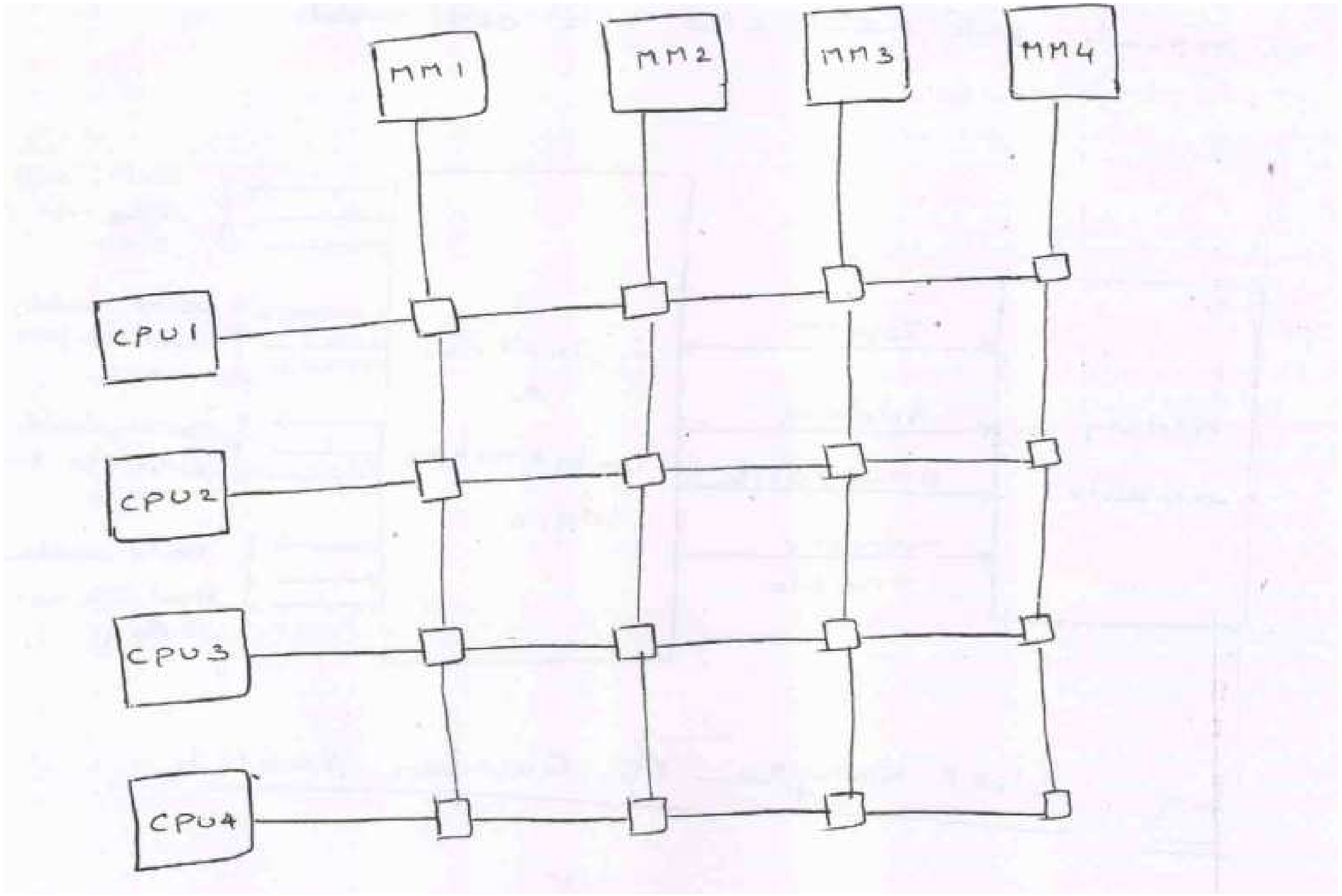
•- High contention

## Shared memory interconnects

- Switched interconnect
  - Uses switches to control the routing of data among the connected devices.

  - Crossbar –
    - Allows simultaneous communication among different devices.
    - Faster than buses.
    - But the cost of the switches and links is relatively high.

# Cross Bar Switch Network



## Figure 2.7

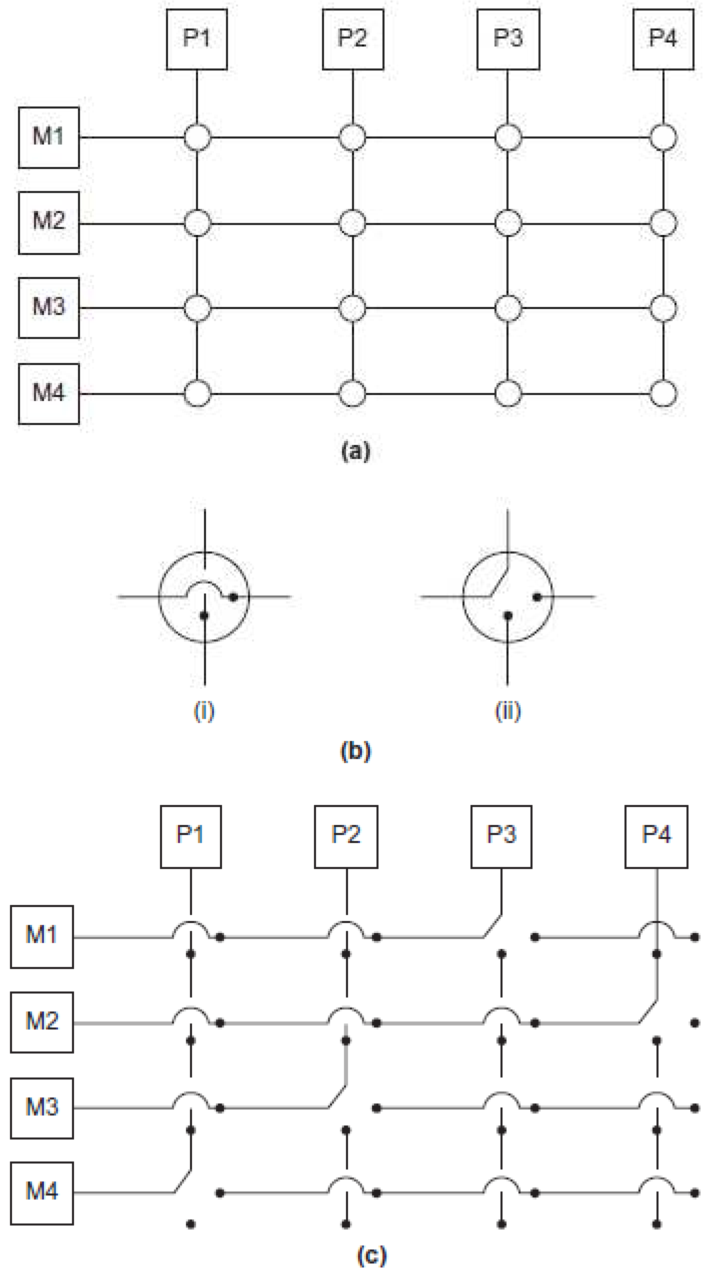
(a)

A crossbar switch connecting 4 processors ($P_i$) and 4 memory modules ($M_j$)

(b)

Configuration of internal switches in a crossbar

(c) Simultaneous memory accesses by the processors



## Cross bar switch network

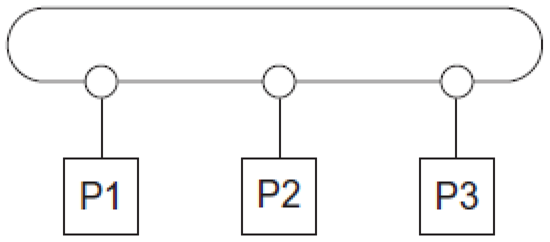
- Every node connected to all others (non-blocking)
- Good for small number of nodes
- Low latency O(1)
- High throughput
- Expensive O(N square2 ) cost
- Difficult to arbitrate

- Ex:IBM POWER5 - Sun Niagara I/II

## Distributed memory interconnects

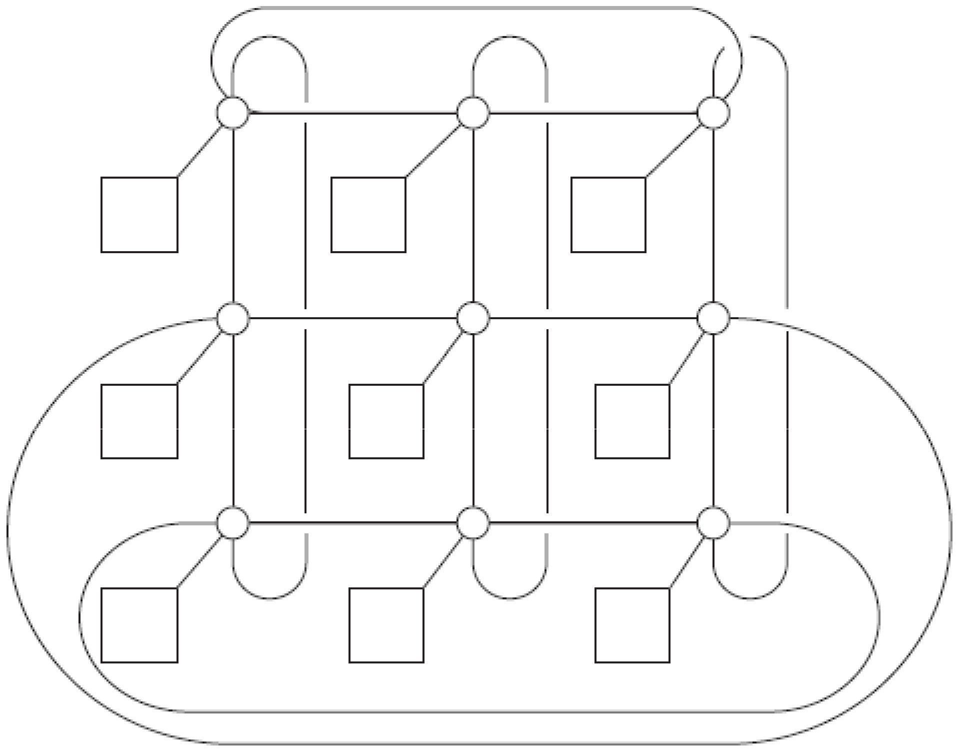
- Two groups
  - Direct interconnect
    - Each switch is directly connected to a processor memory pair, and the switches are connected to each other.
  - Indirect interconnect
    - Switches may not be directly connected to a processor.

# Direct interconnect



(a)

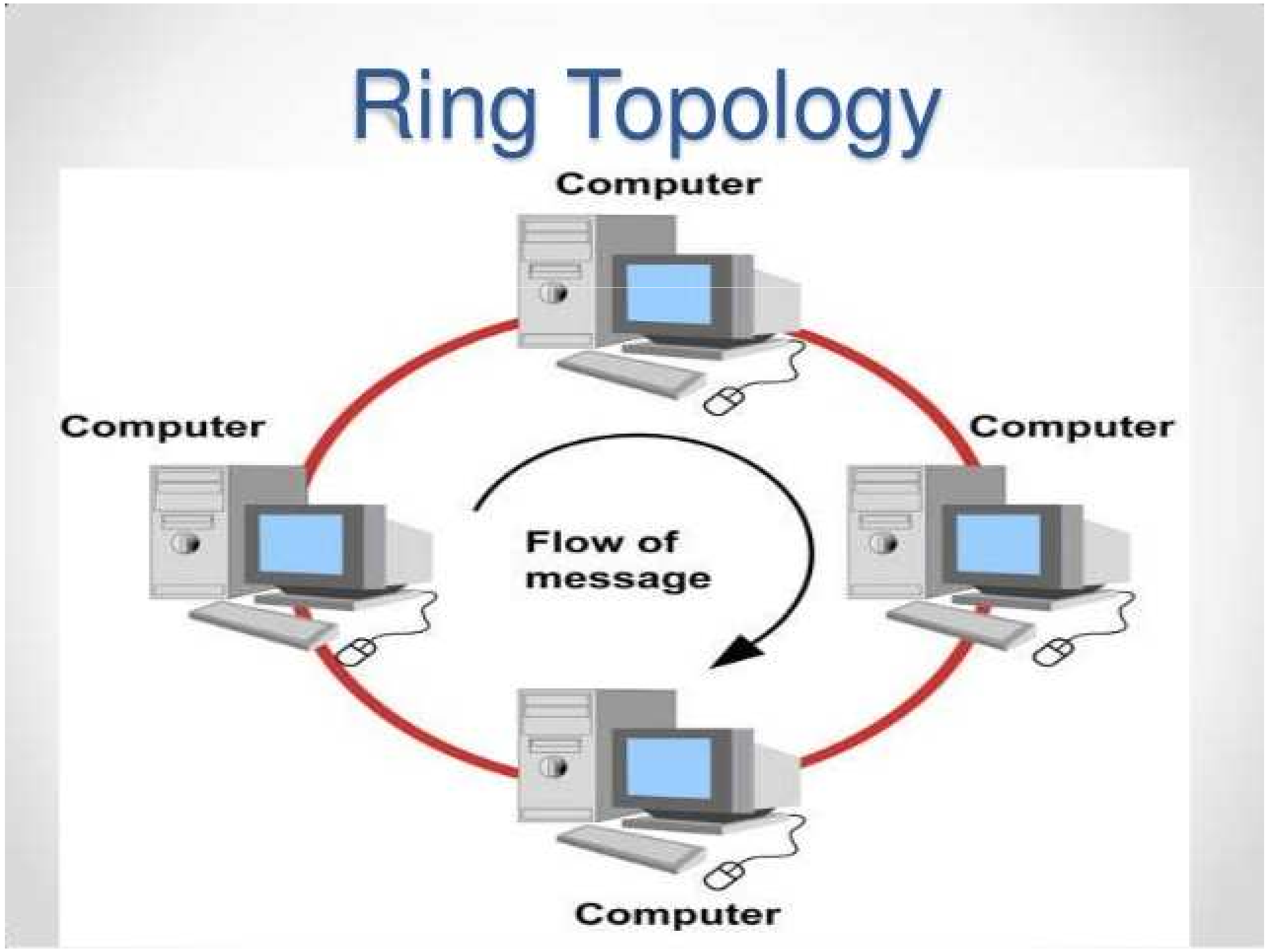
ring



(b)

toroidal mesh

## Ring Network



## Ring Network

- each node is connected to form a single closed data path
- Data from one node is passed along to the next node If that node is not the intended destination, then the data is transmitted to the next node until the destination is reached.
- A token (a special bit pattern) is circulated in the network to enable a node to capture the data.
- Used in: - Intel Larrabee/Core i7 - IBM Cell

## Advantages of Ring Topology

- The ability to achieve transmission rates on the order of 10 million bits per second
- Provision of local communication via a single channel
- Cheap O(N) cost
- No central server (which reduces the cost)

## Disadvantages of Ring Topology

- High latency O(N)
- Not easy to scale
- Failure of one node results in failure of the entire network
- Detection of a fault is very difficult
- Isolation of a fault is not easy

## Mesh Topology

- In which all the network nodes are individually connected to most of the other nodes.
- There is not a concept of a central switch, hub or computer which acts as a central point of communication to pass on the messages.
- It can be divided into two kinds:
  - Fully connected mesh topology
  - Partially connected mesh topology

## Mesh Topology

- fully connected mesh has all the nodes connected to every other node.
- partially connected mesh does not have all the nodes connected to each other.
- Used in: - Tilera 100-core CMP
  - On-chip network prototypes

## Mesh Topology:Advantages

- Each connection can carry its own data load
- It is robust
- A fault is diagnosed easily
- Provides security and privacy
- A broken node won't distract the transmission of data in a mesh network.
- Each node is connected to several other nodes which make it easier to relay data.
- Additional devices in a mesh topology will not affect its network connection
- Easy to layout on-chip: regular & equal-length links
- Path diversity: many ways to get from one node to another

## Mesh Topology:Disadvantages

- There are high chances of redundancy in many of the network connections.
- Overall cost of this network is way too high as compared to other network topologies.
- Set-up and maintenance of this topology is very difficult.
- Administration of the network is tough
- O(N) cost
- Average latency: O(sqrt(N))

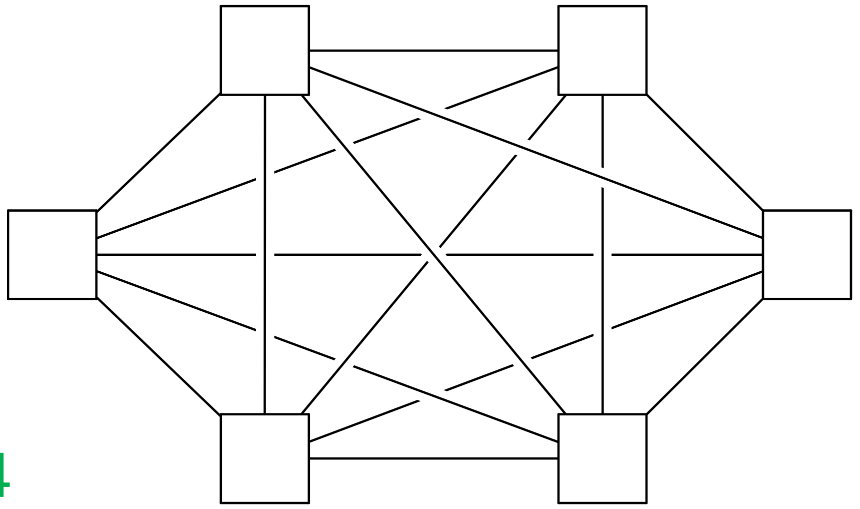
## Definitions

- Bandwidth
  - The rate at which a link can transmit data.
  - Usually given in megabits or megabytes per second.
- Bisection bandwidth
  - A measure of network quality.
  - Instead of counting the number of links joining the halves, it sums the bandwidth of the links.

# Fully connected network

- Each switch is directly connected to every other switch.

impractical



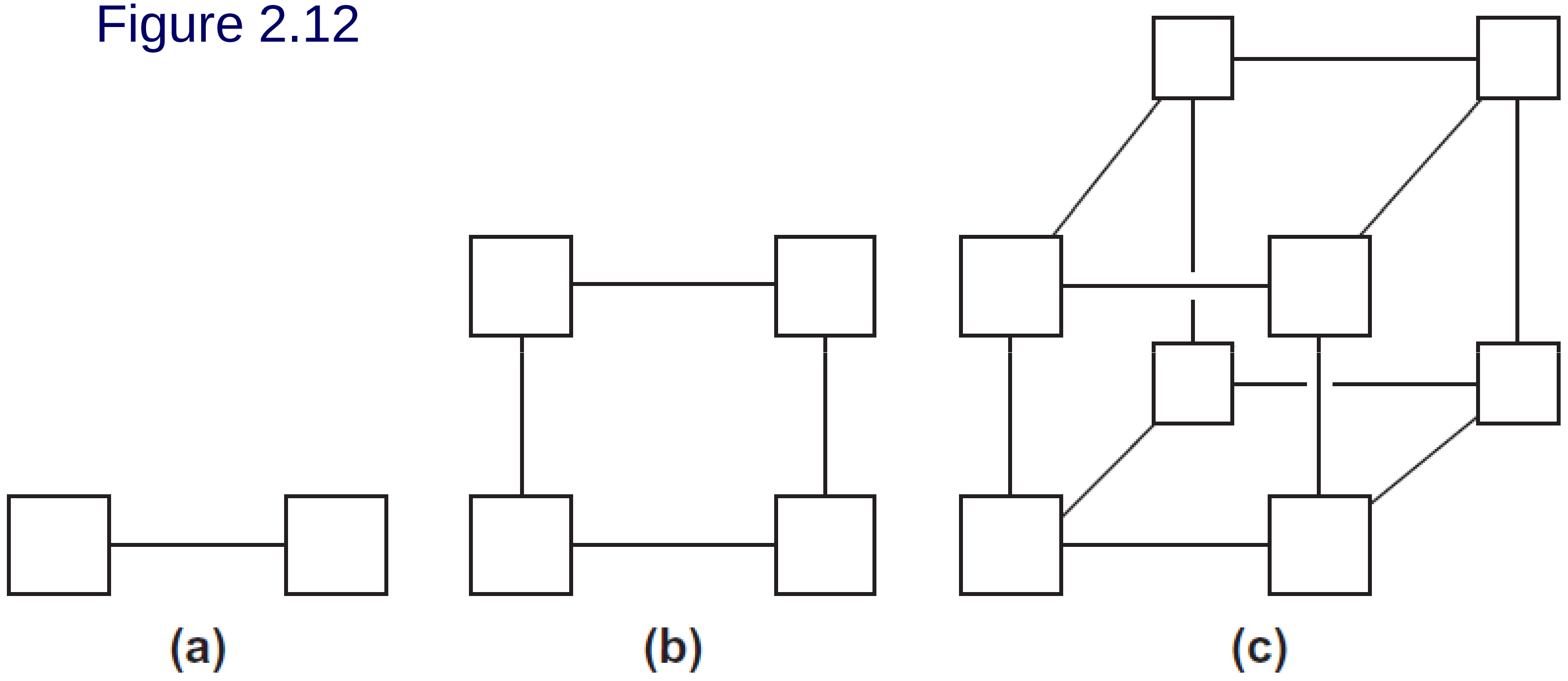
bisection width = $p^2/4$

## Hypercube

- Highly connected direct interconnect.
- Built inductively:
  - A one-dimensional hypercube is a fully-connected system with two processors.
  - A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches.
  - Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.

# Hypercubes

Figure 2.12



one- two- three-dimensional

## Hypercubes

•Binary n-Cube network

•Used in Distributed memory systems

•Has N=2 power n nodes in the network.

•n→dimension of the cube.

•Nodes are assigned (2 power n) n-bit binary addresses

•The addresses are assigned such that address of the neighboring nodes differ in one bit position

## Hypercubes

•Low latency  O(logN)

•Hard to lay out in 2D/3D

• Used in some early message passing machines

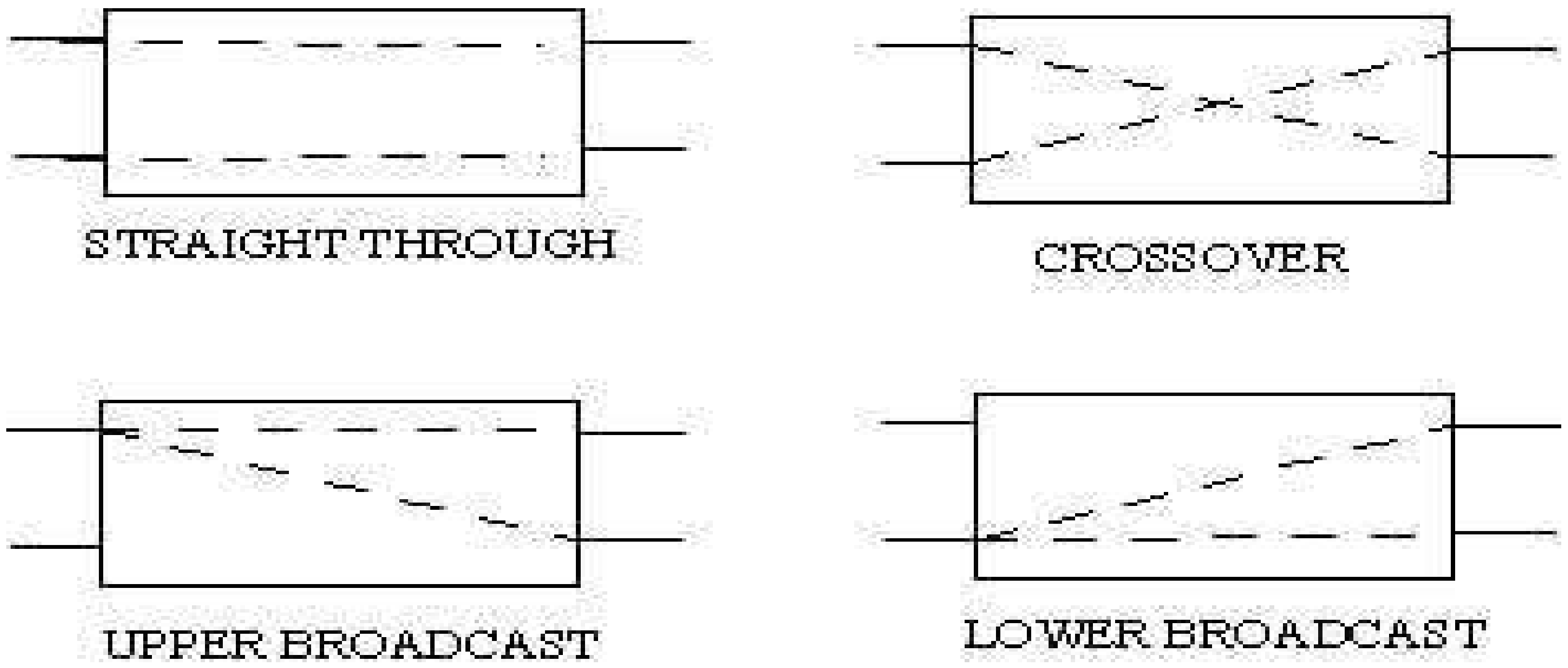
• e.g.: - Intel iPSC, nCube

## Indirect interconnects

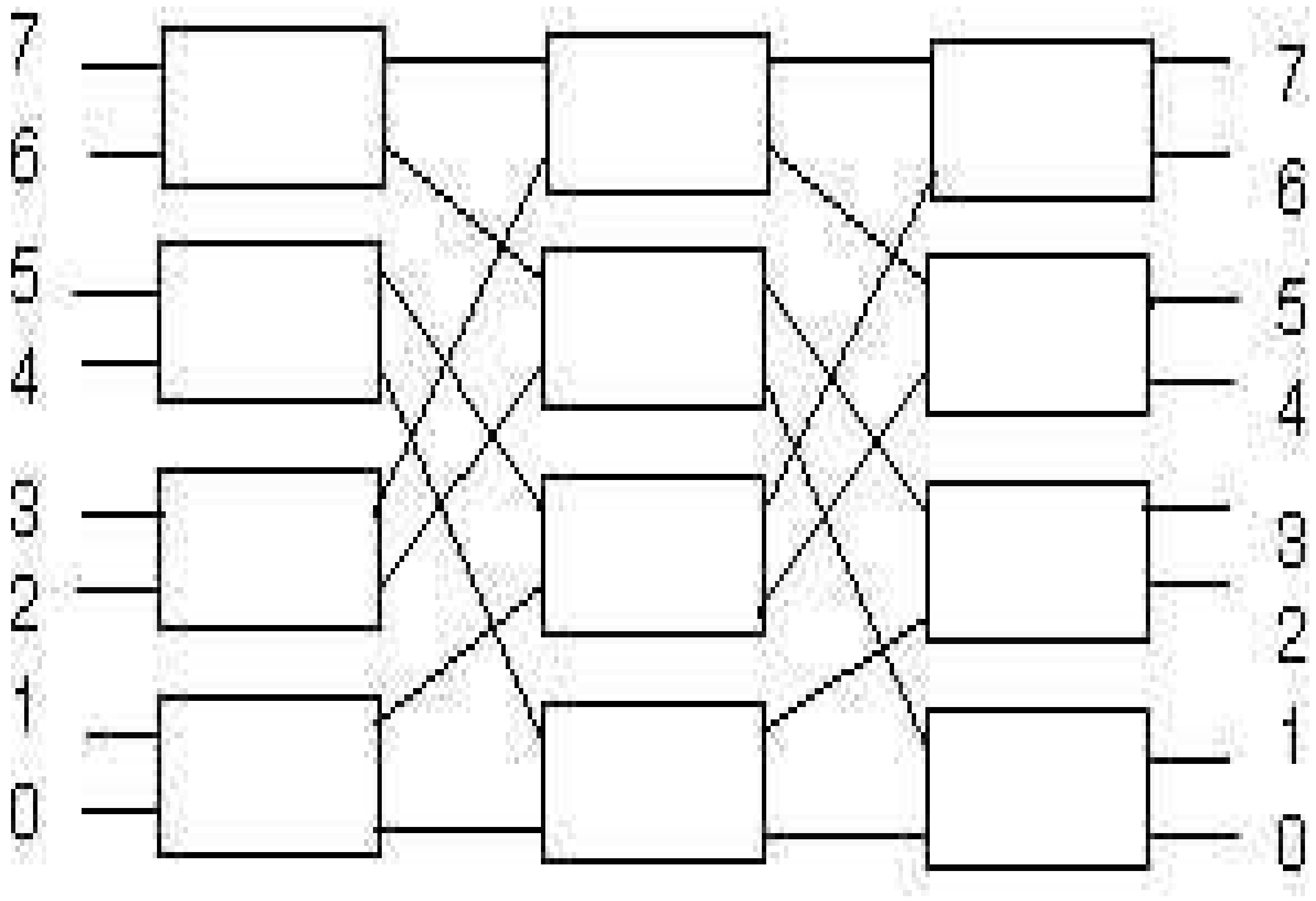
- Simple examples of indirect networks:
  - Crossbar
  - Omega network
- Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.

# Omega network

- The basic element of Omega network is 2x2 switch which has the following possible settings:



# Omega network



# Omega network

- Indirect networks with multiple layers of switches between terminals
- complete connectivity for a set of inputs and outputs is the Omega Network.
-  2 power N nodes using N stages, with each stage containing 2power (N-1) switches.

## Omege Network Disadvantages

- The Omega network is a blocking network because some messages may be blocked by the settings required for other messages.
- It is not fault tolerant if a single switch fails, some connections are no longer possible.
- By adding additional stages, we may increase fault tolerance and reduce or eliminate blocking
- Cost O(Nlog N)
- Latency Olog N)

# More definitions

- Any time data is transmitted, we’re interested in how long it will take for the data to reach its destination.
- Latency
  - The time that elapses between the source’s beginning to transmit the data and the destination’s starting to receive the first byte.
- Bandwidth
  - The rate at which the destination receives data after it has started to receive the first byte.

Message transmission time = l + n / b

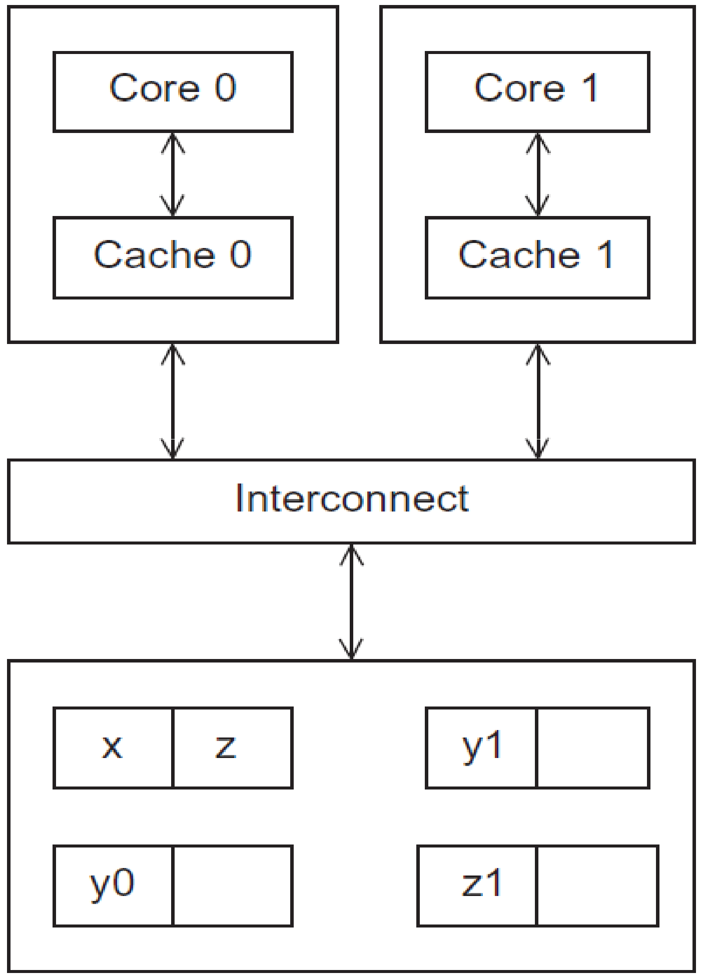
latency (seconds)

length of message (bytes)

bandwidth (bytes per second)

## Cache coherence

- Programmers have no control over caches and when they get updated.



A shared memory system with two cores and two caches

# Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

| Time | Core 0 | Core 1 |
|---|---|---|
| 0 | `y0 = x;` | `y1 = 3*x;` |
| 1 | `x = 7;` | Statement(s) not involving x |
| 2 | Statement(s) not involving x | `z1 = 4*x;` |

y0 eventually ends up = 2

y1 eventually ends up = 6

z1 = ???

## Snooping Cache Coherence

- The cores share a bus .
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.

## Directory Based Cache Coherence

- Uses a data structure called a directory that stores the status of each cache line.

- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated.

## PERFORMANCE

# Speedup

- Number of cores = p
- Serial run-time = $T_{serial}$
- Parallel run-time = $T_{parallel}$

linear speedup

$$T_{parallel} = T_{serial} \;/\; p$$

## Speedup of a parallel program

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

# Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\dfrac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

## Amdahl’s Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

## Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.