

# IT801-N

## Distributed and Parallel Computing

**Book:** *Parallel Programming - Techniques and applications Using Networked Workstations and Parallel Computer*

**Author:** Barry Wilkinson and Michael Allen

**Publisher:** Prentice Hall



**Kadi Sarva Vishwavidyalaya**  
**Faculty of Engineering & Technology**  
**Fourth Year Bachelor of Engineering (IT)**  
(To be Proposed For: Academic Year 2020-21)

<b>Subject Code: IT801-N</b>	<b>Subject Title: Distributed and Parallel Computing</b>
<b>Pre-requisite</b>	C, C++, Java Programming, Computer Organization

**Teaching Scheme (Credits and Hours)**

Teaching scheme				Total Credit	Evaluation Scheme					
L	T	P	Total		Theory		Mid Sem Exam	CIA	Pract.	Total
Hrs	Hrs	Hrs	Hrs		Hrs	Marks	Marks	Marks	Marks	Marks
03	00	02	05	04	03	70	30	20	30	150

**Course Objective:**

- To learn the advanced concepts of Parallel and Distributed Computing and its implementation for assessment of understanding the course by the students.
- Understand the distributed and parallel computing systems.
- Familiar with parallel and distributed languages MPI, Pthread, and OpenMP
- Design parallel and distributed algorithms using these parallel languages

## Detailed Syllabus

Sr. No	Topic	Lecture Hours	Weightage (%)
1	<b>Introduction:</b> The Demand for Computational Speed, Types of Parallel Computers, Cluster Computing	3	6
2	<b>Message Passing Computing:</b> Basics of Message-Passing Programming, Using a Cluster of Computers, Debugging and Evaluating Parallel Programs Empirically	5	10
3	<b>Partitioning and Divide-and-Conquer Strategies:</b> Partitioning, Partitioning and Divide-and-Conquer Examples	5	10
4	<b>Pipelined Computations:</b> Pipeline Technique, Computing Platform for Pipelined Applications, Pipeline Program Examples	7	15
5	<b>Synchronous Computations:</b> Synchronization, Synchronized Computations, Synchronous Iteration Program Examples, Partially Synchronous Methods	8	17
6	<b>Load Balancing and Termination Detection:</b> Load Balancing, Dynamic Load Balancing, Distributed Termination Detection Algorithms, Program Example	6	12
7	<b>Programming with Shared Memory:</b> Shared Memory Multiprocessors, Sharing Data, Parallel Programming Languages and Constructs, OpenMP, Performance Issues	7	15
8	<b>Distributed Shared Memory Systems and Programming:</b> Distributed Shared Memory, Implementing Distributed Shared Memory, Distributed Shared Memory Programming Primitives	7	15
	<b>Total</b>	<b>48</b>	<b>100</b>

# 1 Introduction

[Weightage(6%): Approx. 4-5 Marks out of 70 Marks]

- The Demand for Computational Speed
- Types of Parallel Computers
- Cluster Computing

# 1.1 The Demand for Computational Speed

- Need for more computational power
- Examples of high computational demand
  - Simulations in engineering and engineering applications
  - Weather forecasting
  - Modelling DNA structure
- Weather Forecasting
  - Atmosphere can be modelled in 3-dimensions
  - Factors: Temperature, pressure, humidity, wind speed, wind direction etc.
  - Computations are made over a specific time intervals

# 1.1 The Demand for Computational Speed

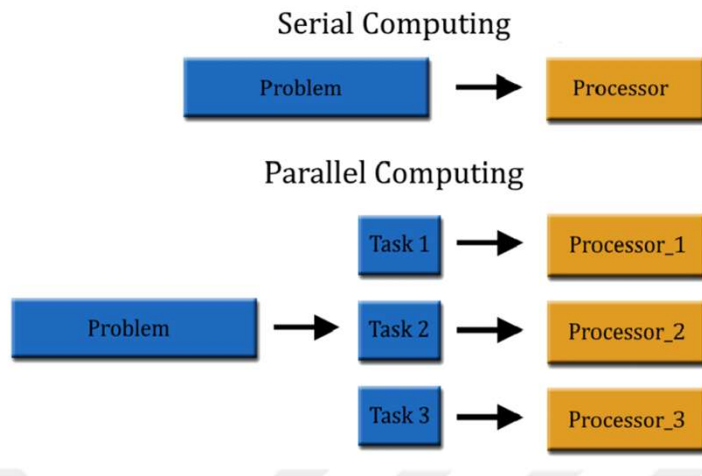
- Weather Forecasting
  - Sample Statistics:
    - Cell size: 1 mile x 1 mile x 1 mile [Height 10 miles, 10 cells high]
    - $5 \times 10^8$  cells
    - 200 FLOPs per cell
    - FLOPs:  $5 \times 10^8 \times 200 = 10^{11}$  FLOPS
    - Time interval: 1 min, Forecasting for: 7 days: 60 min/hour x 24 hours/day x 7 days = 10080 ( $10^4$ ) time stamps
    - Total FLOPs:  $10^{11} \times 10^4 = 10^{15}$
    - Core i5 (with quad core) can perform 20 GFLOPs per second
    - Time required:  $10^{15} / 20 \times 10^9 = 50,000$  seconds [ $\approx 14$  hours]

# 1.1 The Demand for Computational Speed

- Predicting the motion of astronomical bodies in the space
  - Sample Statistics:
    - Each body is attracted by other bodies by gravitational forces
    - Movement of the body is determined by total forces applicable on it
    - If there are total  $N$  bodies, there will be  $N-1$  forces applicable on individual body
    - For all  $N$  bodies, approximate total  $N^2$  forces are to be computed
    - Once a motion of a body is determined, it is to be recomputed over a sample period, continuously.
    - There are total ( $N=$ )  $10^{11}$  stars in a galaxy, for instance.
    - Hence, total calculations are ( $N^2=$ )  $10^{22}$
    - Assuming the complexity of an efficient algorithm:  $N \log_2 N = 10^{11} \log_2 10^{11} = 36.11 \times 10^{11}$  operations per second
    - Core i5 (with quad core) can perform 20 GFLOPs per second
    - Time required:  $36.11 \times 10^{11} / 20 \times 10^9 = 180$  times slower

# 1.1 The Demand for Computational Speed

- Looking at both the examples (weather forecasting and motion prediction), we conclude that there is a need of more computational power, which can be achieved through (i) multiple processors within a single system (multiprocessor) or (ii) multiple computers working on a single problem.
- In both cases, the problem is split into parts and each part is performed on separate processor. All such processors work in parallel.
- Such computing model could be known as *Parallel Computing*. And writing programs for such platform is called *Parallel Programming*.

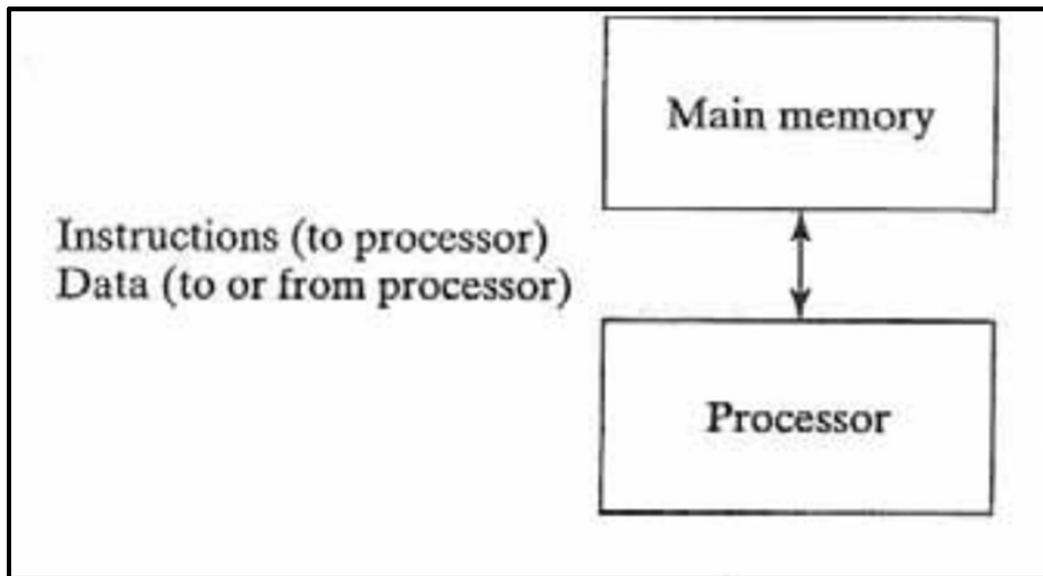


- Ideally, using  $p$  processor would result into achievement of  $p$  times computational speed and would get the result in  $1/p$  time.
- Practically, there are issues of (i) problems can not be divided perfectly into parts (ii) interaction among the parts is required (synchronization).

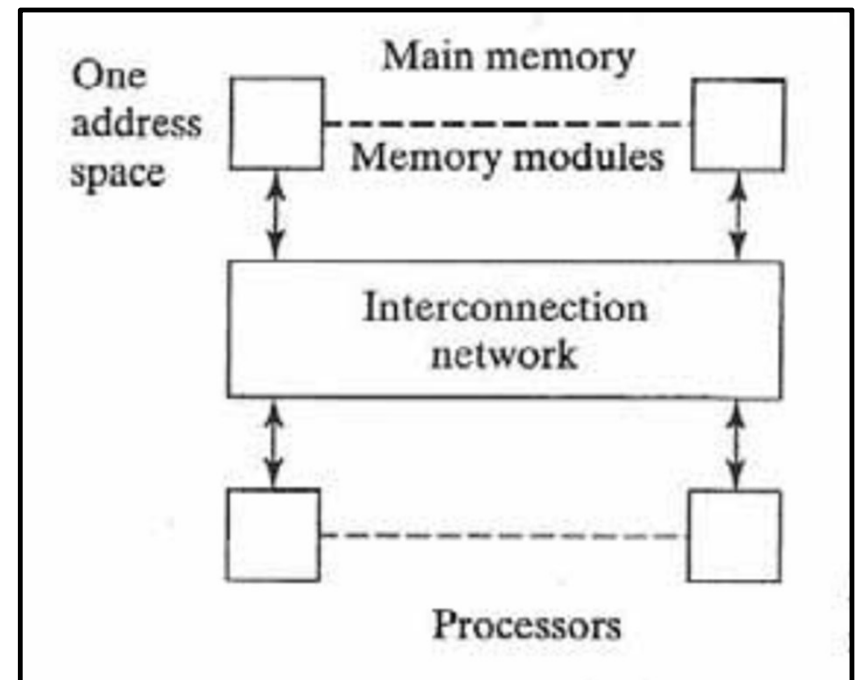


## 1.2 Types of Parallel Computers

- Shared memory multiprocessor
- Distributed-memory multicomputer



Conventional computer having a single processor and memory

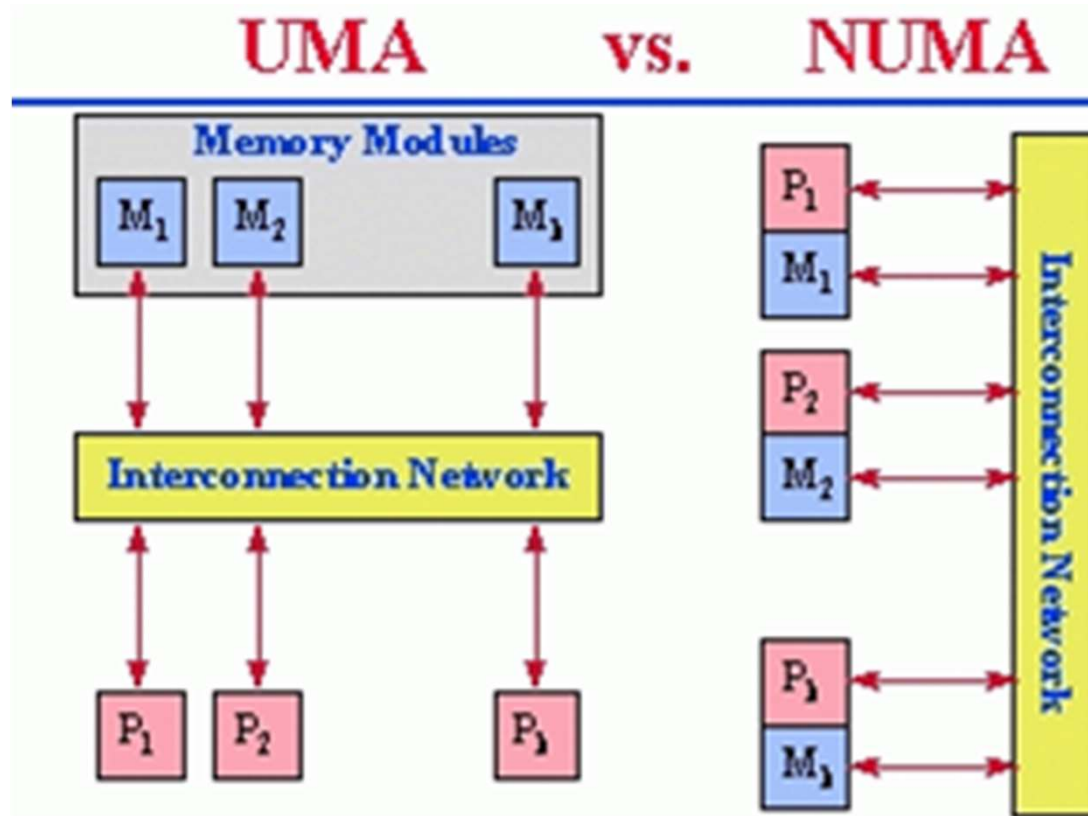


Traditional shared memory multiprocessor model

## 1.2 Types of Parallel Computers

- Programming a Shared memory multiprocessor
  - Code and data (for each program) are stored in the shared memory
  - Typically, all the processors execute the same program.
  - Need for high level parallel programming language
  - Compiler produces the final executable code
  - Sometimes, a regular **sequential** programming language would be used with *pre-processor directives* to specify the parallelism. (E.g **OpenMP**)
  - Alternatively, *threads* can be used
  - One more way is to use regular **sequential** programming language and modify the *syntax* to specify parallelism (E.g. **UPC, Unified Parallel C**)

# 1.2 Types of Parallel Computers



## Uniform Memory Access (UMA):

In UMA, where Single memory controller is used. UMA is slower than NUMA. In UMA, bandwidth is restricted or limited rather than NUMA. There are 3 types of buses used in UMA which are: Single, Multiple and Crossbar. It is applicable for general purpose applications and time-sharing applications.

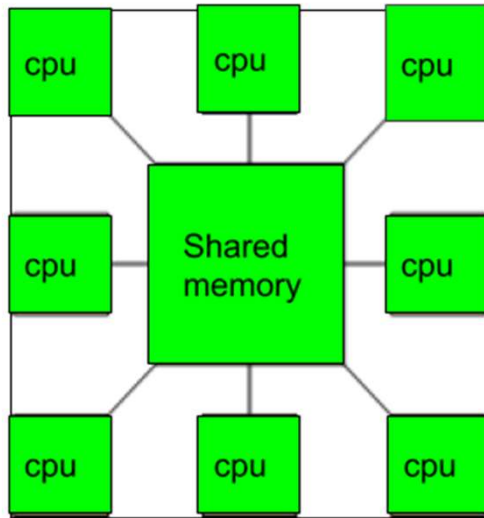
## Non-uniform Memory Access (NUMA):

In NUMA, where different memory controller is used. NUMA is faster than uniform Memory Access. Non-uniform Memory Access is applicable for real-time applications and time-critical applications.

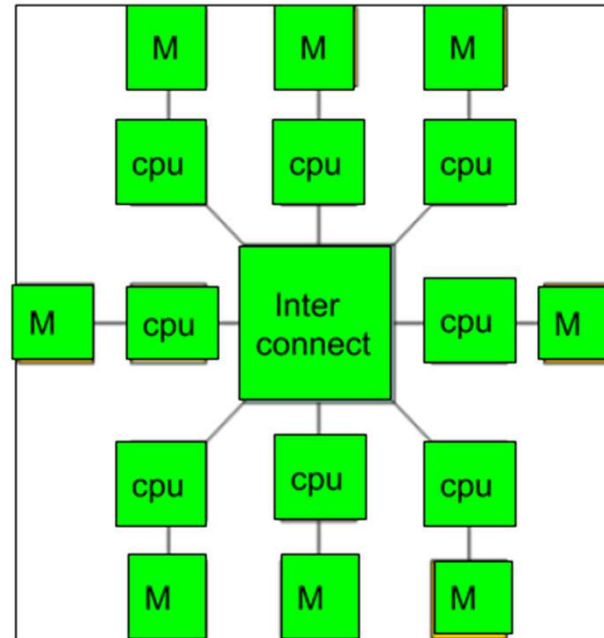
# UMA Vs. NUMA

Sr. No.	Key	UMA	NUMA
1	Definition	Uniform Memory Access.	Non Uniform Memory Access.
2	Memory Controller	Single memory controller	Multiple memory controllers.
3	Memory Access	Slower	Faster
4	Bandwidth	Limited.	More bandwidth.
5	Suitability	Used in general purpose and time sharing applications.	Used in real time and time critical applications.
6	Memory Access time	Has equal memory access time.	Has varying memory access time.
7	Bus types	3 types of Buses supported: Single, Multiple and Crossbar.	2 types of Buses supported: Tree, hierarchical.

# Multiprocessor Vs. Multicomputer

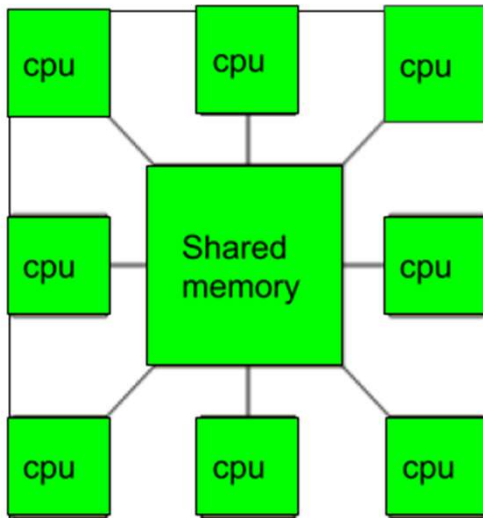


A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM.



A multicomputer system is a computer system with multiple processors that are connected together to solve a problem.

# Multiprocessor System



There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs share the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.

## Applications

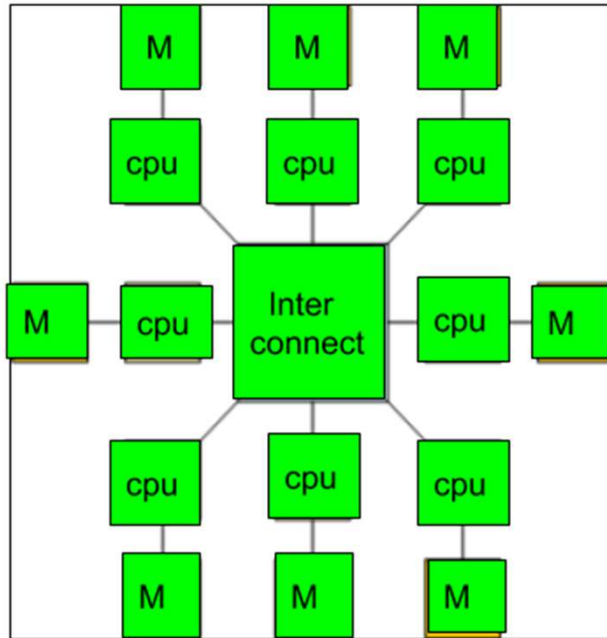
- As a uniprocessor, such as single instruction, single data stream (SISD).
- As a multiprocessor, such as single instruction, multiple data stream (SIMD), which is usually used for vector processing.
- Multiple series of instructions in a single perspective, such as multiple instruction, single data stream (MISD), which is used for describing hyper-threading or pipelined processors.
- Inside a single system for executing multiple, individual series of instructions in multiple perspectives, such as multiple instruction, multiple data stream (MIMD).

## Benefits

- Enhanced performance.
- Multiple applications.
- Multi-tasking inside an application.
- High throughput and responsiveness.
- Hardware sharing among CPUs.

Source: <https://www.geeksforgeeks.org/introduction-of-multiprocessor-and-multicomputer/>

# Multicomputer System



Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network. As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor.

# Multiprocessor Vs. Multicomputer

## Difference between multiprocessor and Multicomputer:

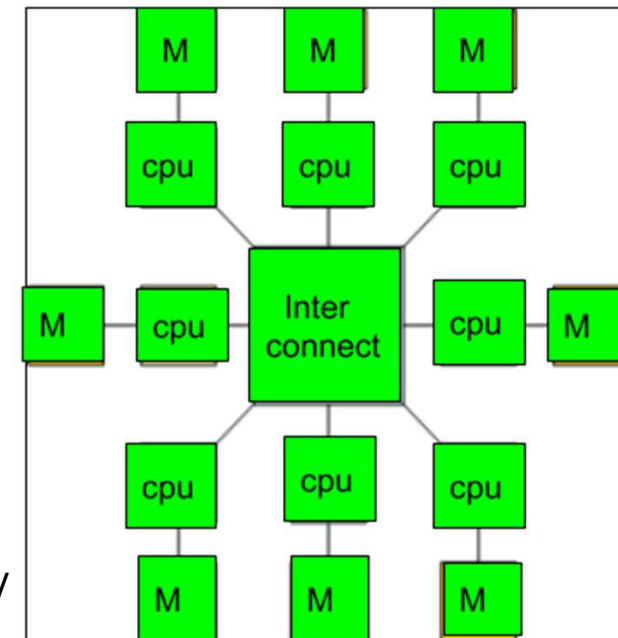
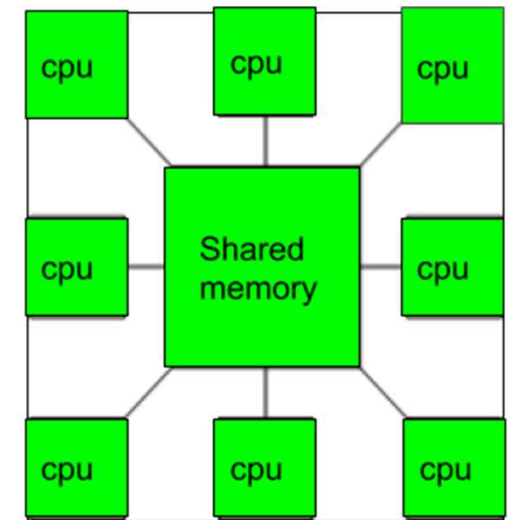
Multiprocessor is a system with two or more central processing units (CPUs) that is capable of performing multiple tasks where as a multicomputer is a system with multiple processors that are attached via an interconnection network to perform a computation task.

A multiprocessor system is a single computer that operates with multiple CPUs where as a multicomputer system is a cluster of computers that operate as a singular computer.

Construction of multicomputer is easier and cost effective than a multiprocessor.

In multiprocessor system, program tends to be easier where as in multicomputer system, program tends to be more difficult.

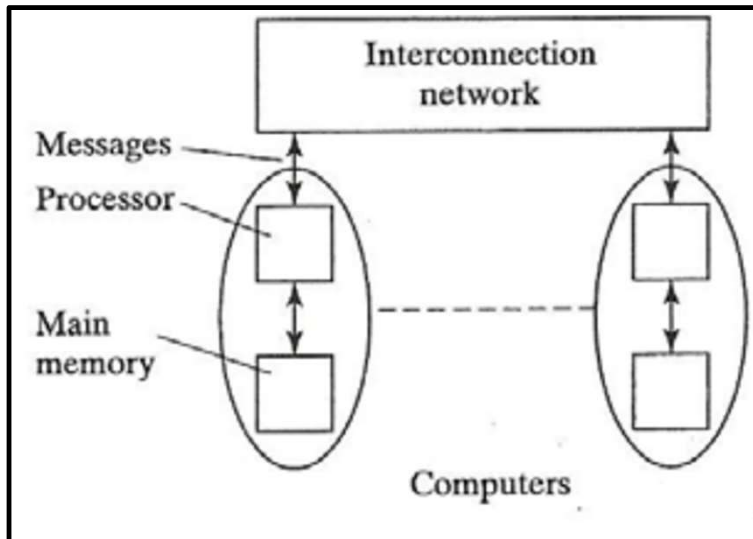
Multiprocessor supports parallel computing, Multicomputer supports distributed computing.



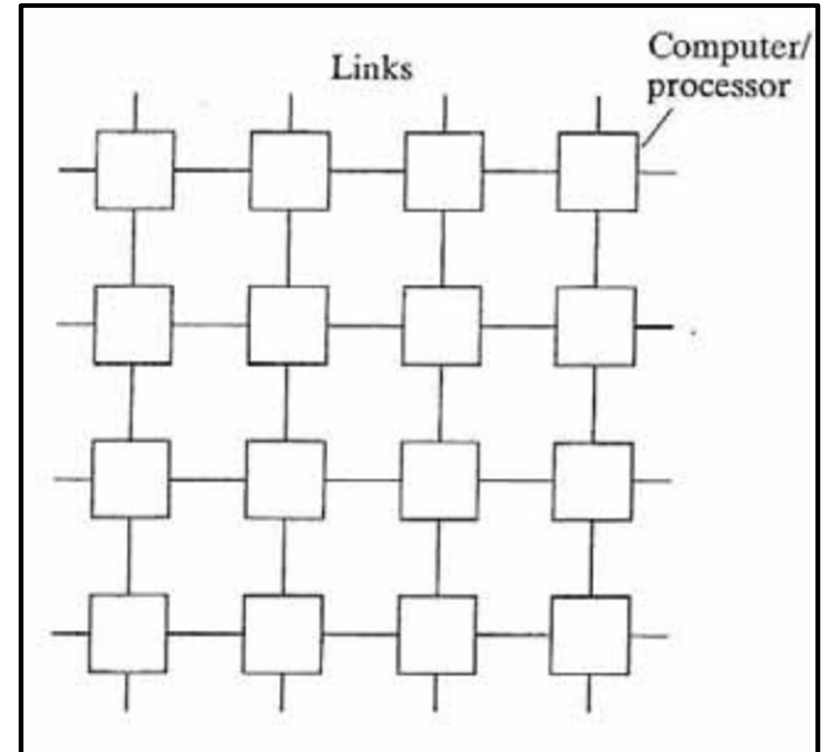
Source: <https://www.geeksforgeeks.org/introduction-of-multiprocessor-and-multicomputer/>



# Message Passing in Multiprocessor/Multicomputer

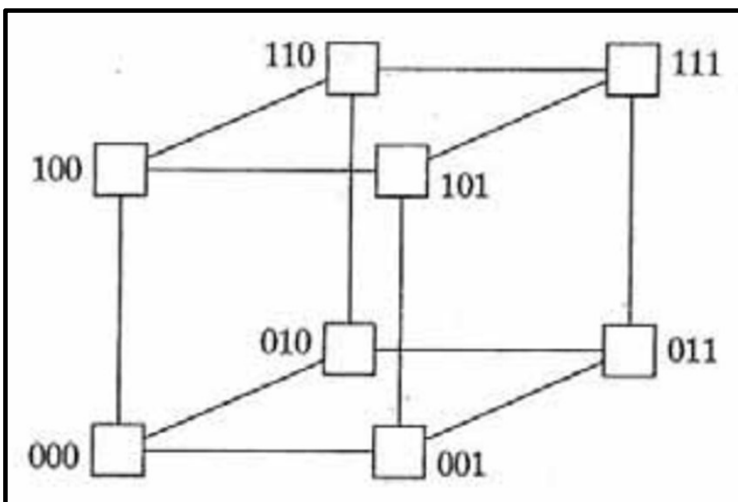


Network of multicomputer systems



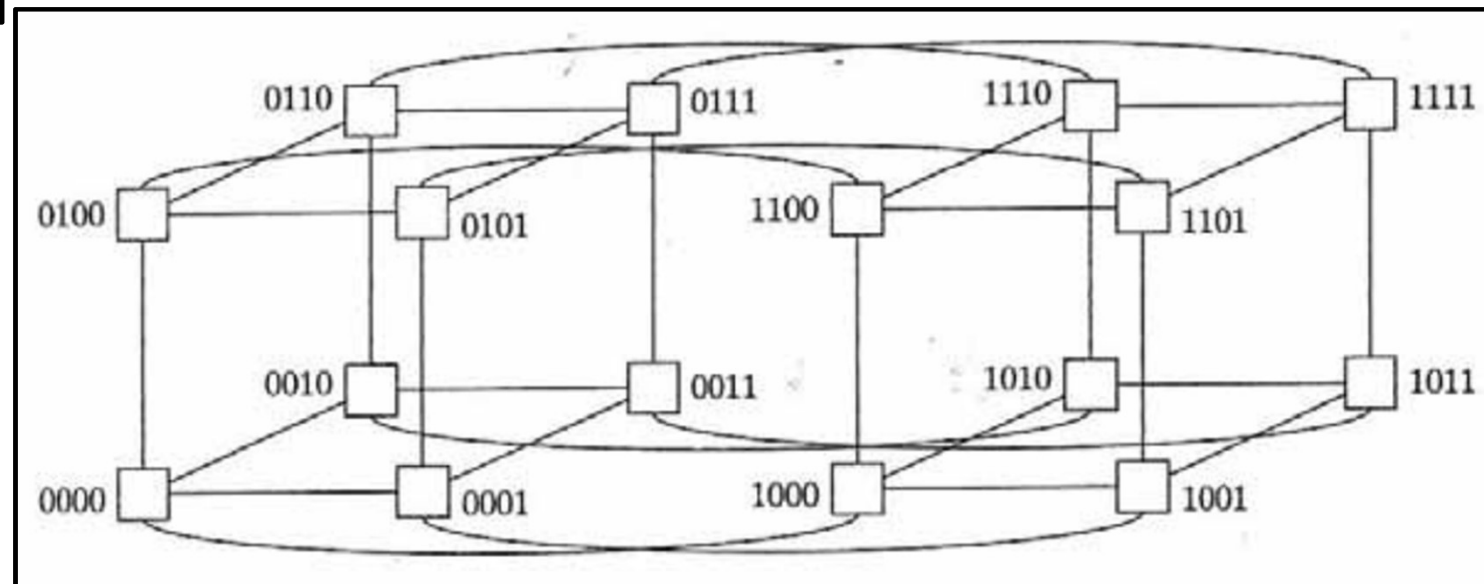
2-D Mesh

# Message Passing in Multiprocessor/Multicomputer

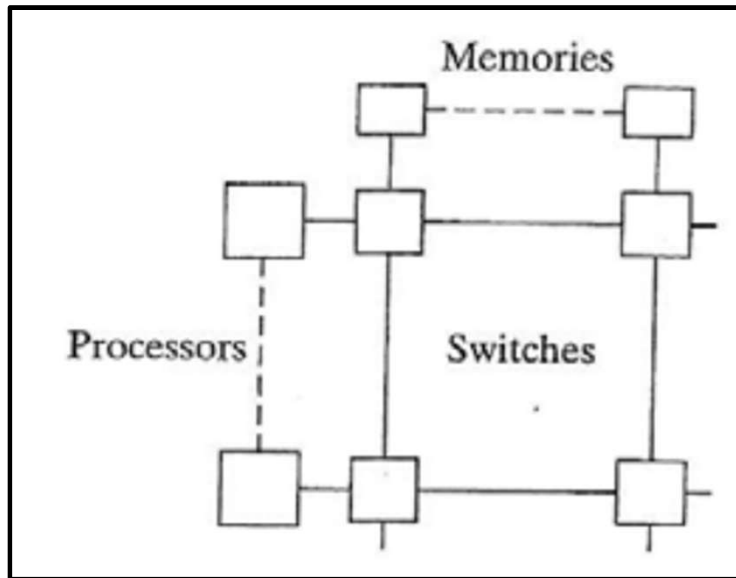


3-D Hypercube

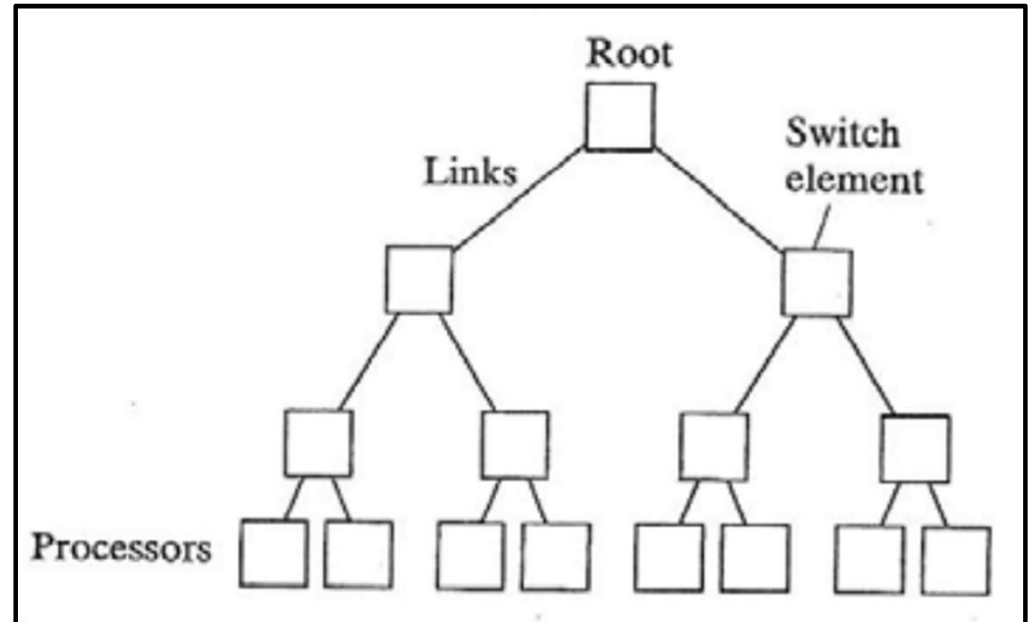
4-D Hypercube



# Message Passing in Multiprocessor/Multicomputer

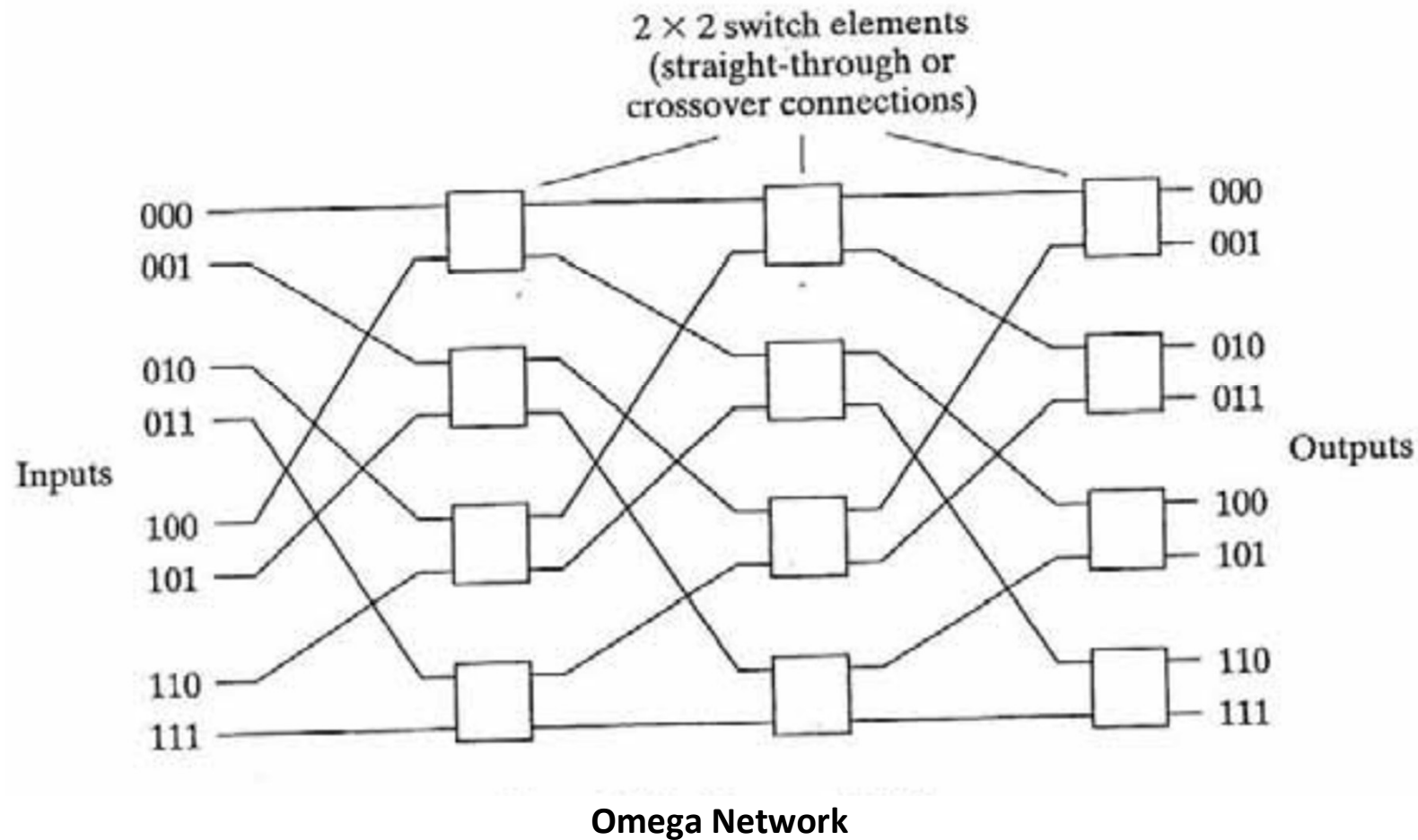


Cross-bar switch

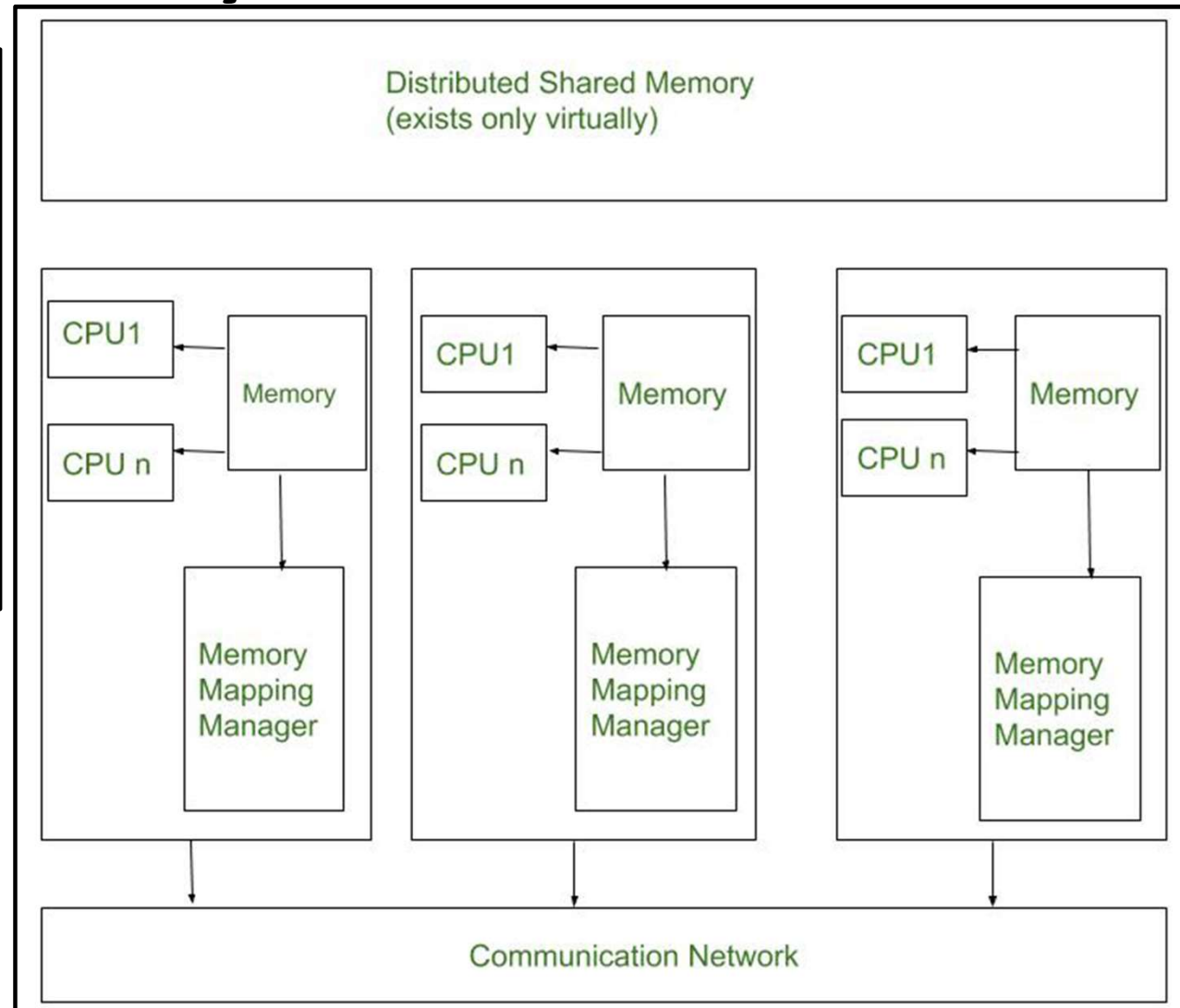
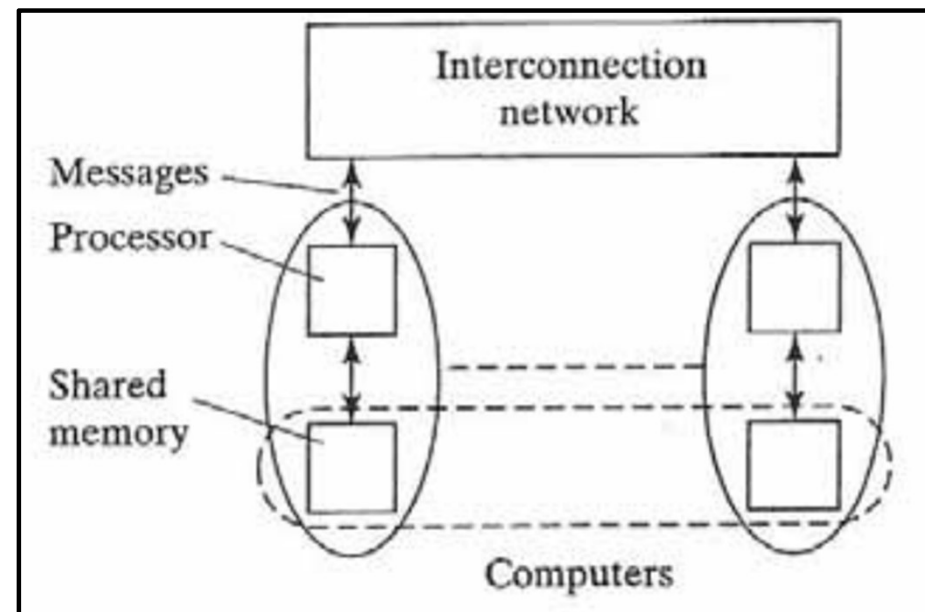


Tree Structure

# Message Passing in Multiprocessor/Multicomputer



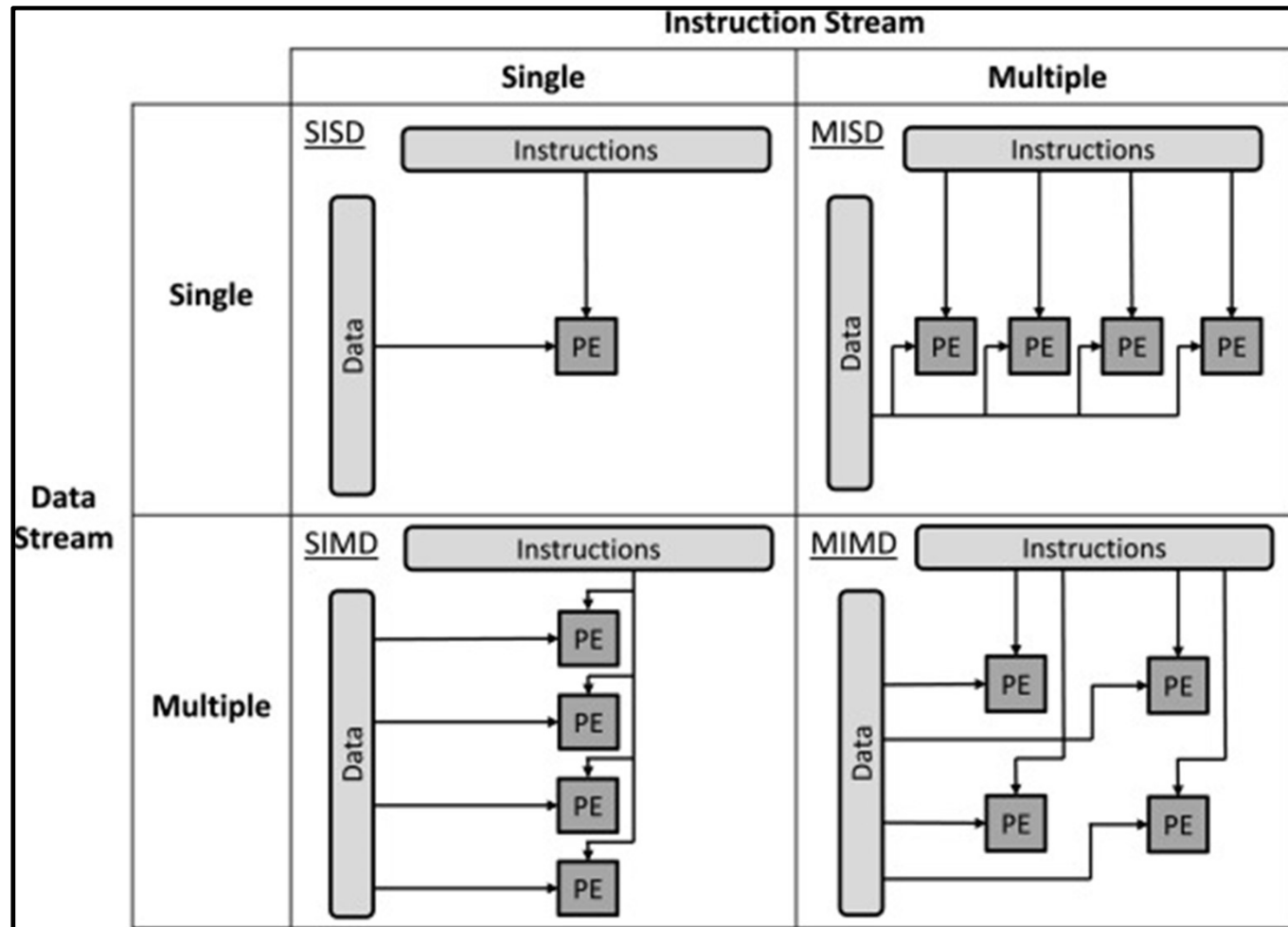
# Distributed Shared Memory



# SIMD Vs MIMD (Flynn's Classification / Taxonomy)

		Instruction Streams	
		one	many
Data Streams	one	<b>SISD</b> traditional von Neumann single CPU computer	<b>MISD</b> May be pipelined Computers
	many	<b>SIMD</b> Vector processors fine grained data Parallel computers	<b>MIMD</b> Multi computers Multiprocessors

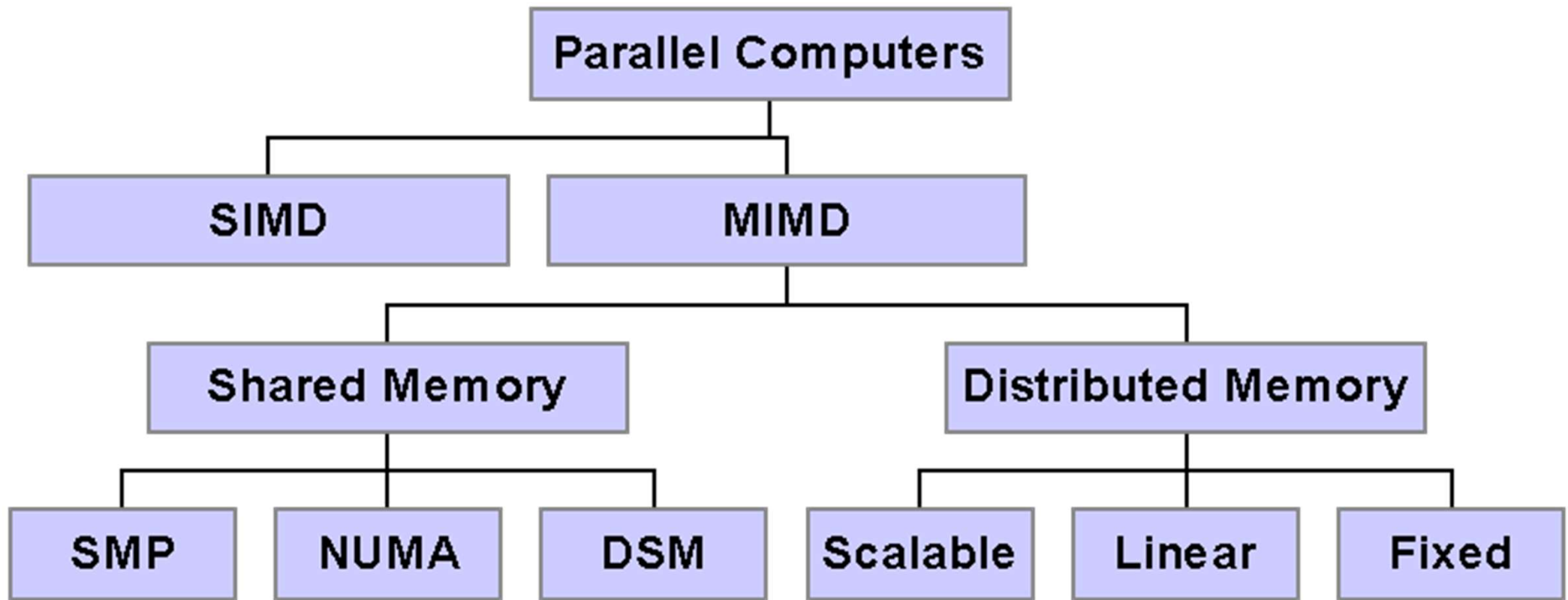
# SIMD Vs MIMD (Flynn's Classification / Taxonomy)



# SIMD Vs MIMD

S.NO	SIMD	MIMD
1.	SIMD stands for Single Instruction Multiple Data.	While MIMD stands for Multiple Instruction Multiple Data.
2.	SIMD requires small or less memory.	While it requires more or large memory.
3.	The cost of SIMD is less than MIMD.	While it is costlier than SIMD.
4.	It has single decoder.	While it have multiple decoders.
5.	It is latent or tacit synchronization.	While it is accurate or explicit synchronization.
6.	SIMD is a synchronous programming.	While MIMD is a asynchronous programming.
7.	SIMD is a simple in terms of complexity than MIMD.	While MIMD is complex in terms of complexity than SIMD.
8.	SIMD is less efficient in terms of performance than MIMD.	While MIMD is more efficient in terms of performance than SIMD.





SIMD: Single Instruction Multiple Data Streams

SIMD: Multiple Instructions Multiple Data Streams

SMP: Symmetrical Multi Processors

NUMA: Non-uniform Memory Access

DSM: Distributed Shared Memory

## 1.3 Cluster Computing

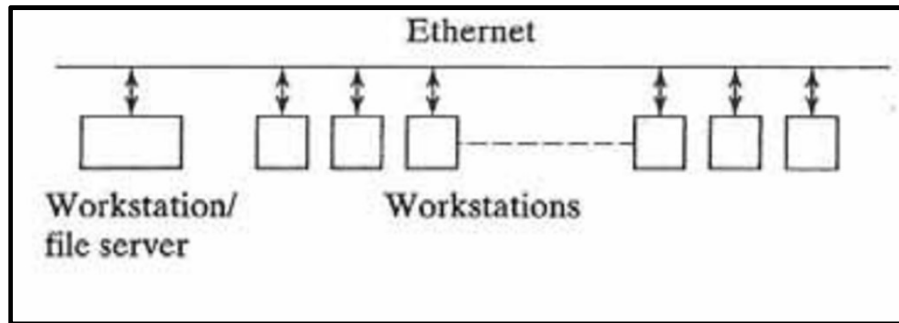
It is a cost-effective approach using WSs/PCs connected together to form powerful computing platforms.

E.g.: Cluster of workstations (COWs), Network of workstations (NOWs)

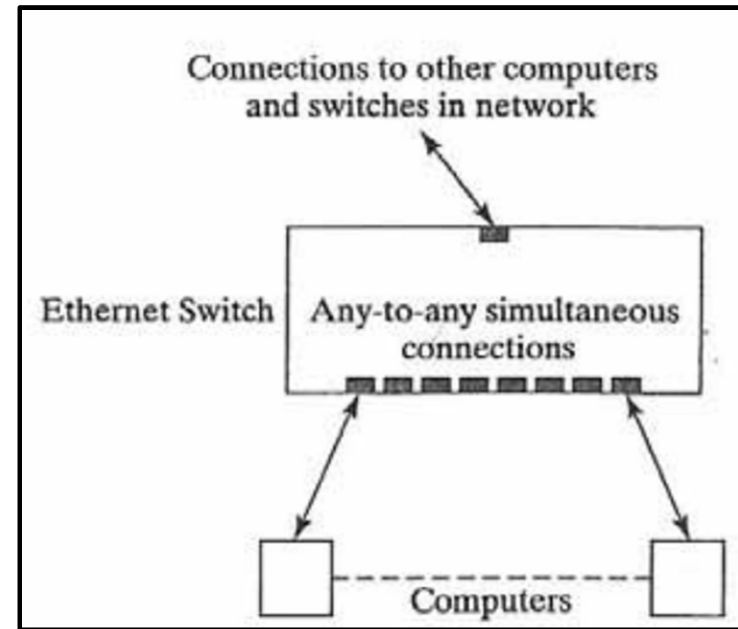
### **Advantages:**

- Very high performance WSs/PCs are readily available at low cost.
- The latest processors can easily be incorporated into the system as they become available and the system can be expanded incrementally by additional computers, disks and other resources.
- Existing application software can be used or modified.

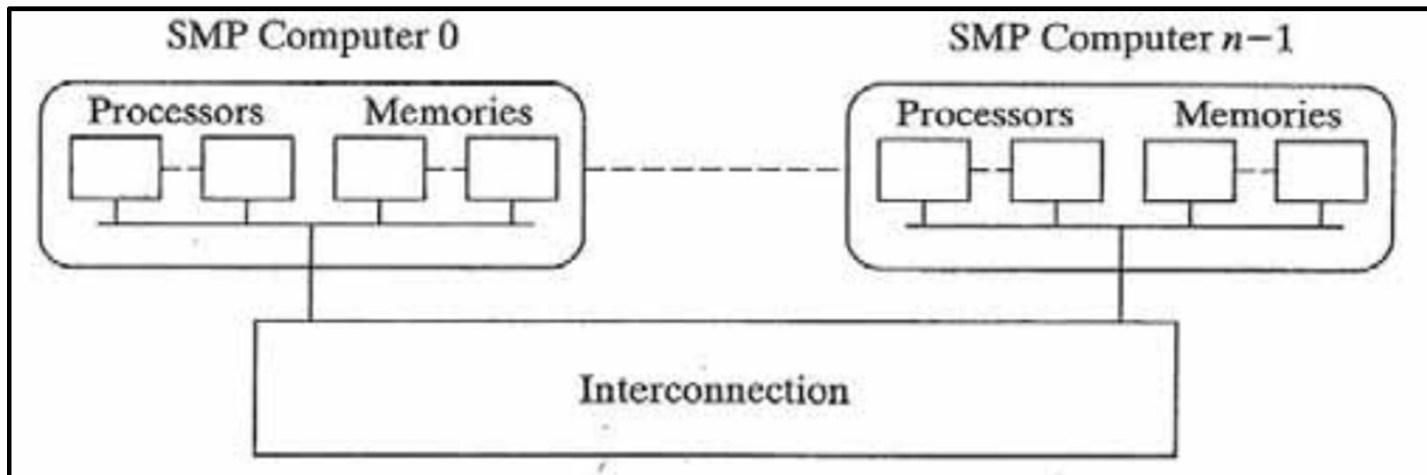
# 1.3 Cluster Computing



Original Ethernet-type single wire network

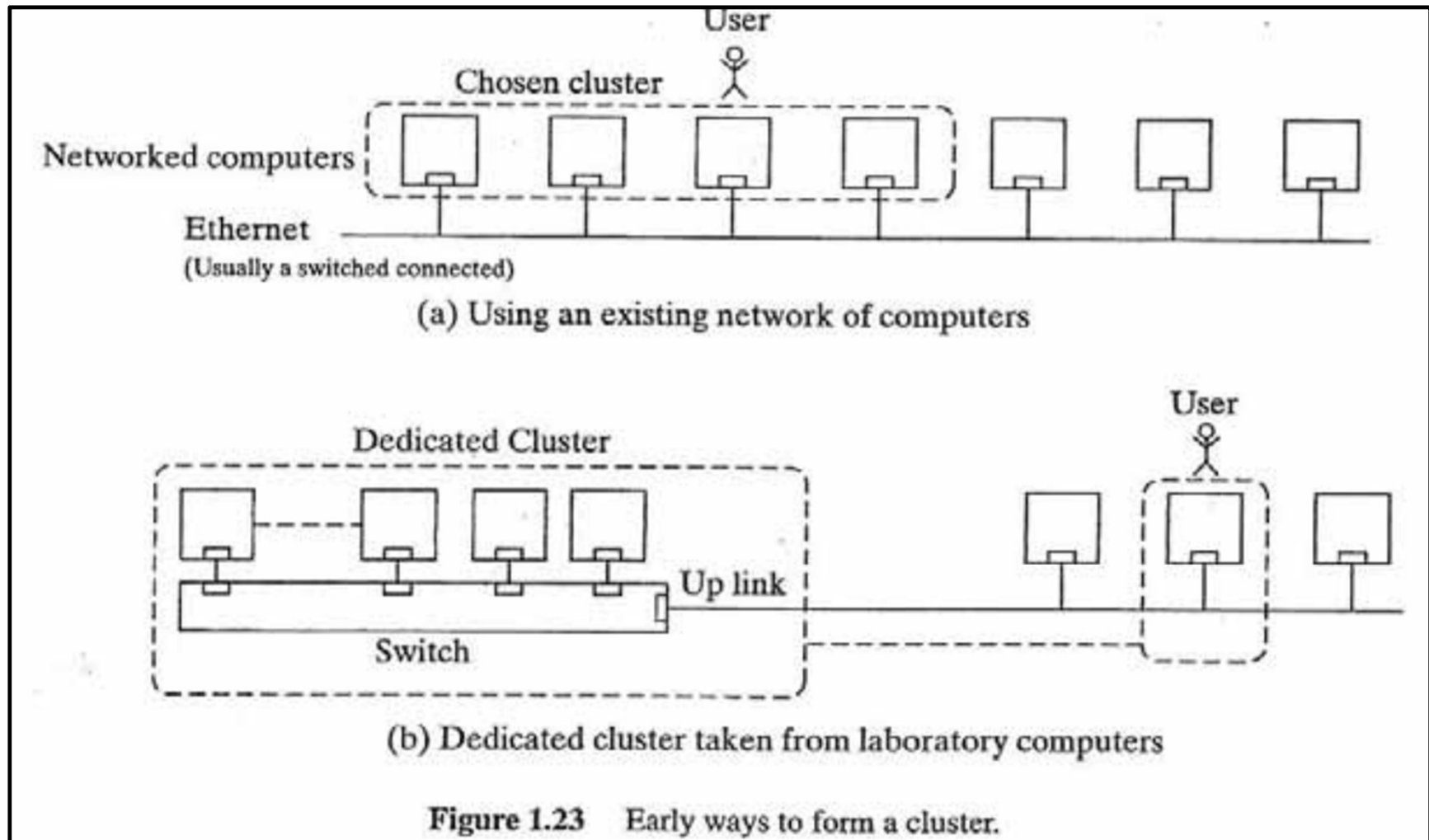


Ethernet Switch



Cluster of Shared Memory Computers

## 1.3 Cluster Computing (Configuration)



# 1 Introduction: Questions

[Weightage(6%): Approx. 4-5 Marks out of 70 Marks]

1. Why do we need more computational power? Explain with any one real-time example.
2. Differentiate UMA Vs. NUMA.
3. Differentiate Multiprocessor Vs. Multicomputer
4. List various topologies used for message passing system. Explain any one of them with neat diagram.
5. Explain message passing multicomputer in detail [5, April 2022]
6. Explain Flynn's classification / taxonomy. OR. Differentiate SIMD Vs. MIMD.
7. Explain Cluster computing.

## 2 Message Passing Computing

[Weightage(10%): Approx. 7 Marks out of 70 Marks]

- Basics of Message-Passing Programming
- Using a Cluster of Computers
- Debugging and Evaluating Parallel Programs Empirically

# 2.1 Basics of Message-Passing Programming

## 2.1.1 Programming Options

- Designing a special **parallel programming** language
  - (E.g.) *occam* designed for *transputer* in 1984
- Extending the *syntax*/reserved words of an existing **sequential language** to handle message-passing
  - (E.g.) *High Performance Fortran (HPF)*
  - (E.g.) *Fortran M*
- Using an existing **sequential language** and providing a *library* of external procedure for message-passing
  - (E.g.) *C*

# 2.1 Basics of Message-Passing Programming

## 2.1.2 Process Creation

- **Static:** Processes are specified before execution (by the programmer). System will execute a fixed number of processes.
- **Dynamic:** Processes can be created and their execution initiated during the running of other processes. It is better than static way of process creation but it incurs significant overhead.

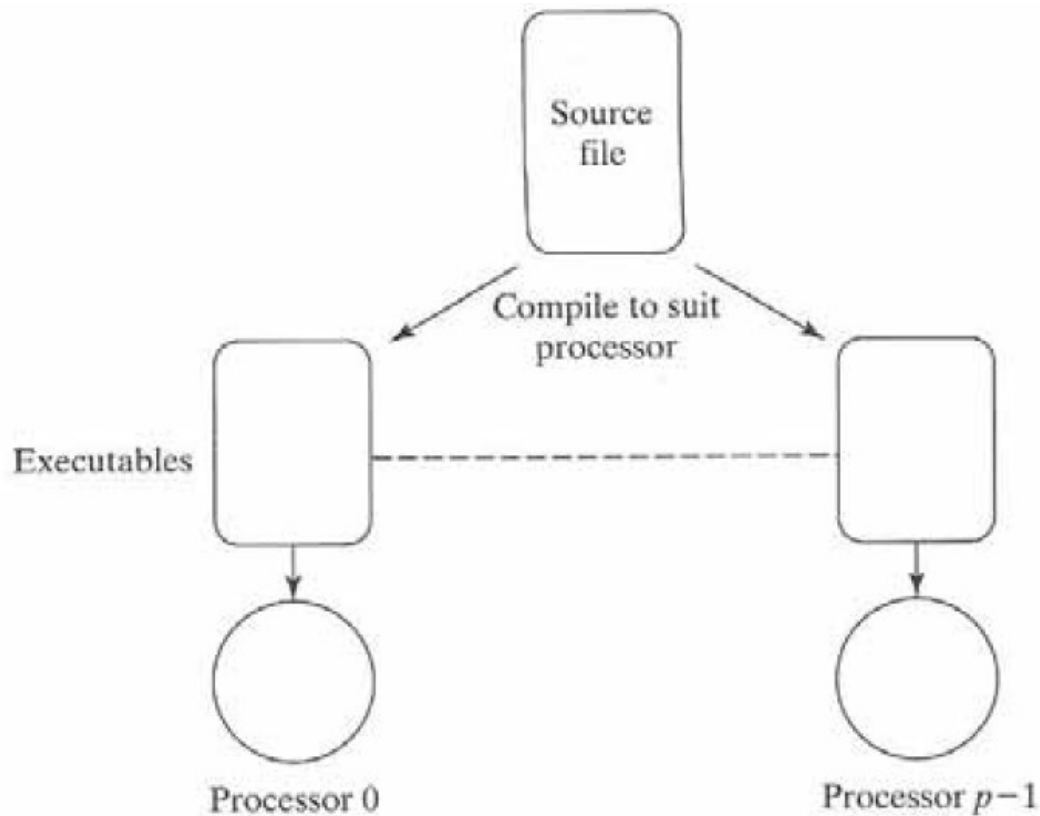
Usually, there is one controlling process, a “master” process, and the reminder are “slaves” or “workers”.

Every process have a unique process ID.

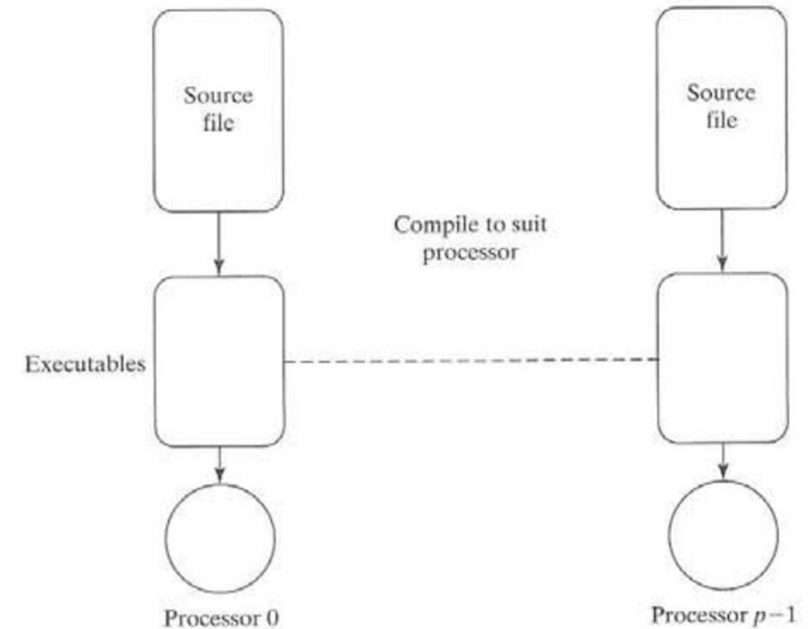


# 2.1 Basics of Message-Passing Programming

## Programming Model



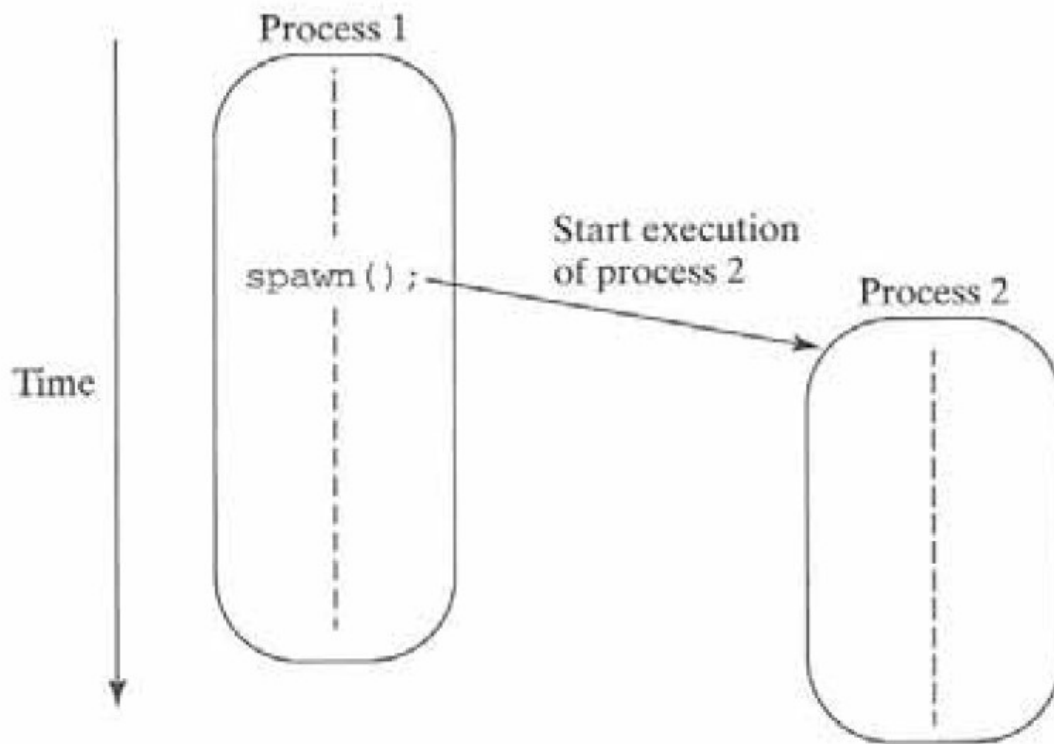
Single-program Multiple-data Model (SPMD)



Multiple-program Multiple-data Model (MPMD)

## 2.1 Basics of Message-Passing Programming

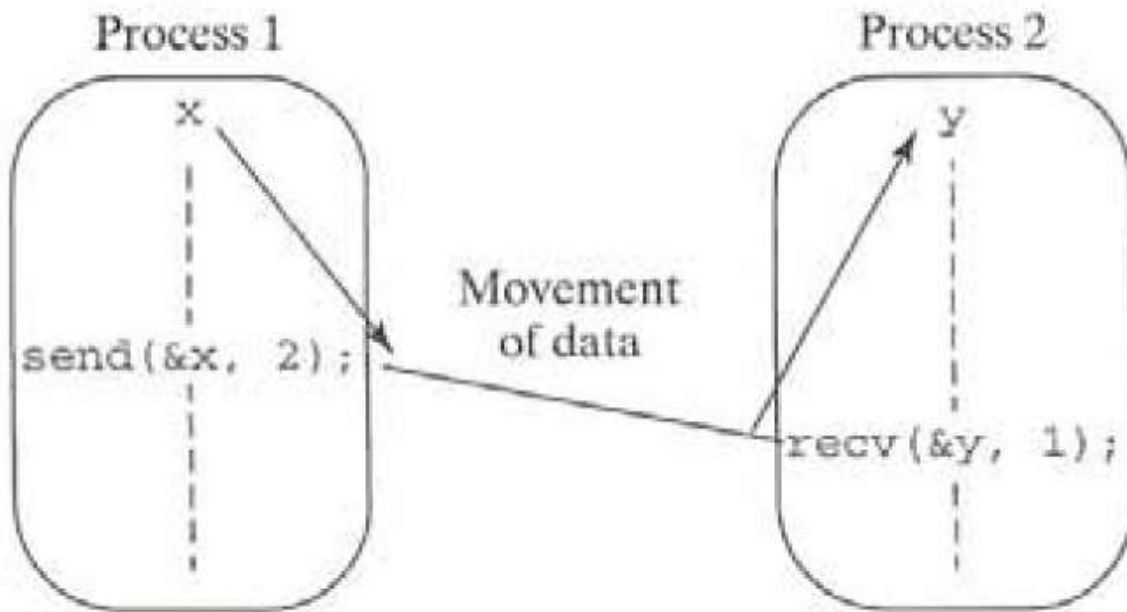
### Spawning a process



For dynamic process creation, two distinct programs (a master and a slave) may be written and compiled. A library call such as `spawn(name_of_process)` starts another process.

# 2.1 Basics of Message-Passing Programming

## Message Passing Routines



Message Passing Library Calls:

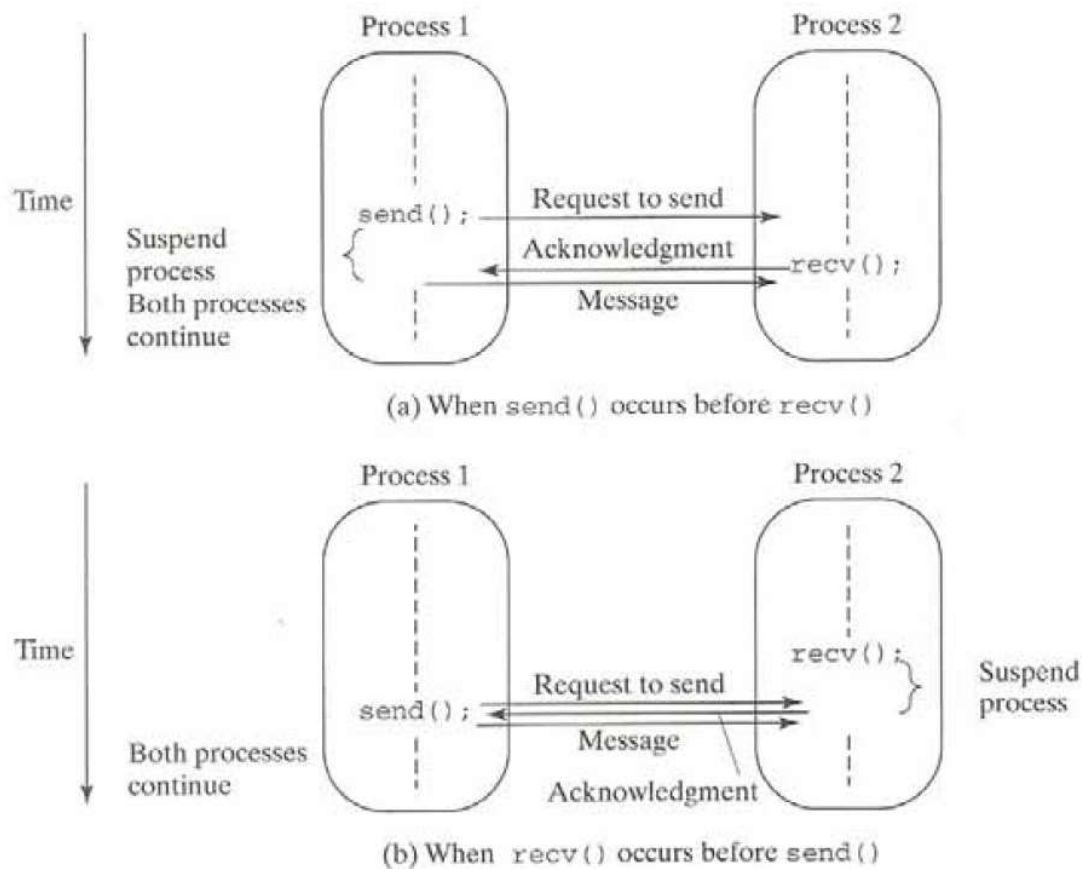
- `send (parameter_list)`
- `recv (parameter_list)`

Typical Example:

```
send (data, dest_process_id)
recv (data, source_process_id)
```

# 2.1 Basics of Message-Passing Programming

## Synchronous Message Passing



The term ***synchronous*** is used for routines that return when the message transfer has been completed.

A ***synchronous send*** routine will wait until the complete message that it has sent has been accepted by the receiving process before returning.

A ***synchronous receive*** routine will wait until the complete message it is expecting arrives and the message is stored before returning.

# 2.1 Basics of Message-Passing Programming

## Blocking and Non-Blocking Message Passing

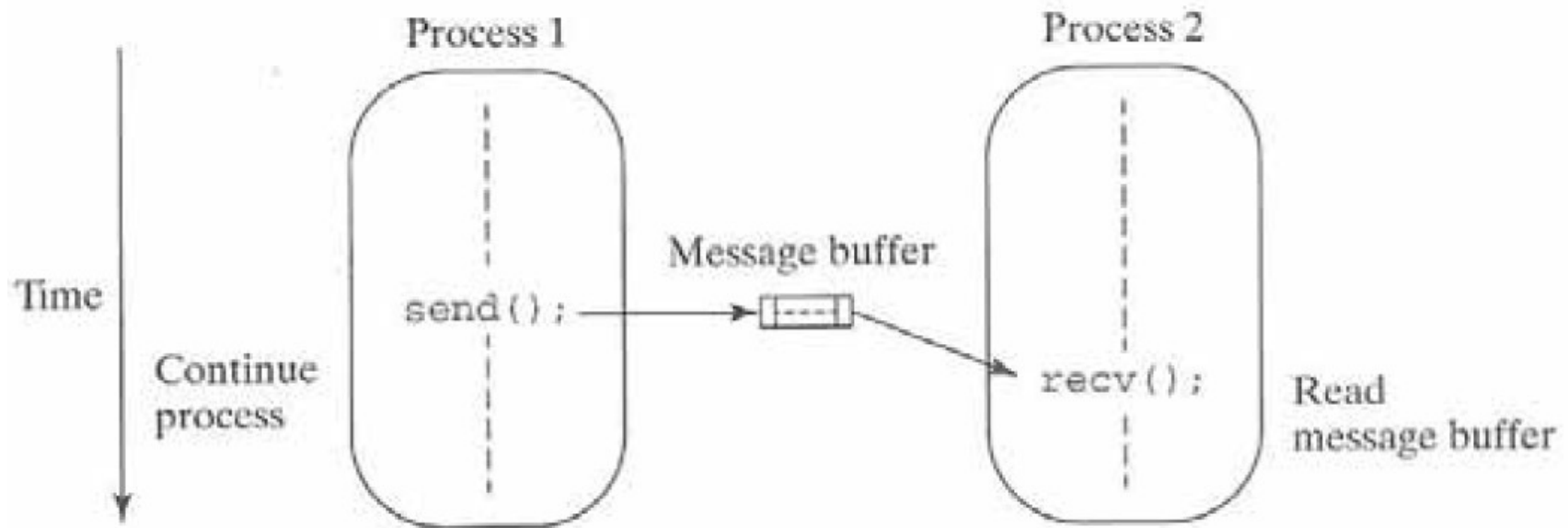
**Blocking:** It describes the routines that do not allow the process to continue until the transfer is completed. That means, the routine is ***blocked*** from continuing.

Here, the terms, Synchronous and Blocking, seem similar.

**Non-blocking:** It describes the routines that return whether or not the message had been received.

# 2.1 Basics of Message-Passing Programming

## Message Buffer



# 2.1 Basics of Message-Passing Programming

## Message Selection

**The normal semantic for message is:**

```
send (message, dest_process_id)
recv (message, source_process_id)
```

A special symbol or number may be provided as a *wild card* in place of `source_process_id` to allow the destination to accept messages from any source.

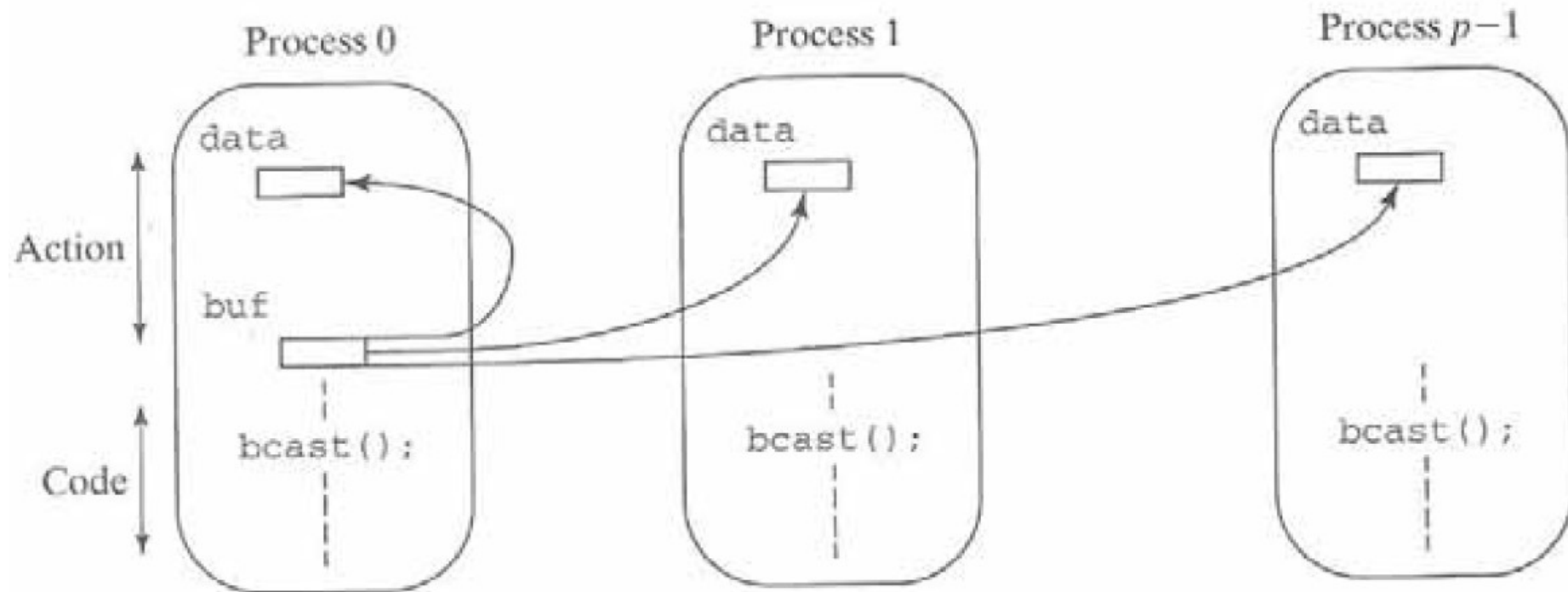
To provide more flexibility, message can be selected by a *message tag* attached to the message. *Message tag* `msgtag` can be is normally a user-chosen integer and it is used to differentiate between different types of messages.

```
send (&x, 2, 5)
send (message, dest_process_id, msgtag)

recv (&y, 1, 5)
send (message, source_process_id, msgtag)
```

## 2.1 Basics of Message-Passing Programming

### Broadcast

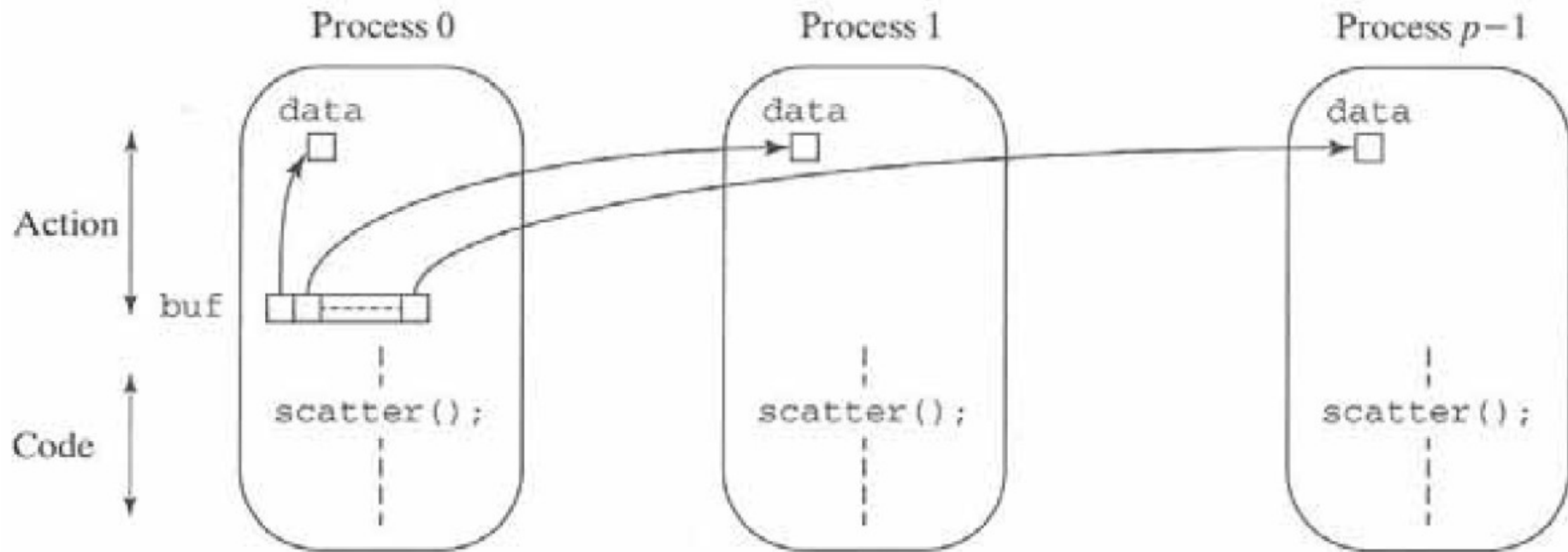


Broadcast: Sending the same message to all the processes concerned. The processes that are to participate must be identified, typically by first forming a named group of processes to be used as a parameter.



## 2.1 Basics of Message-Passing Programming

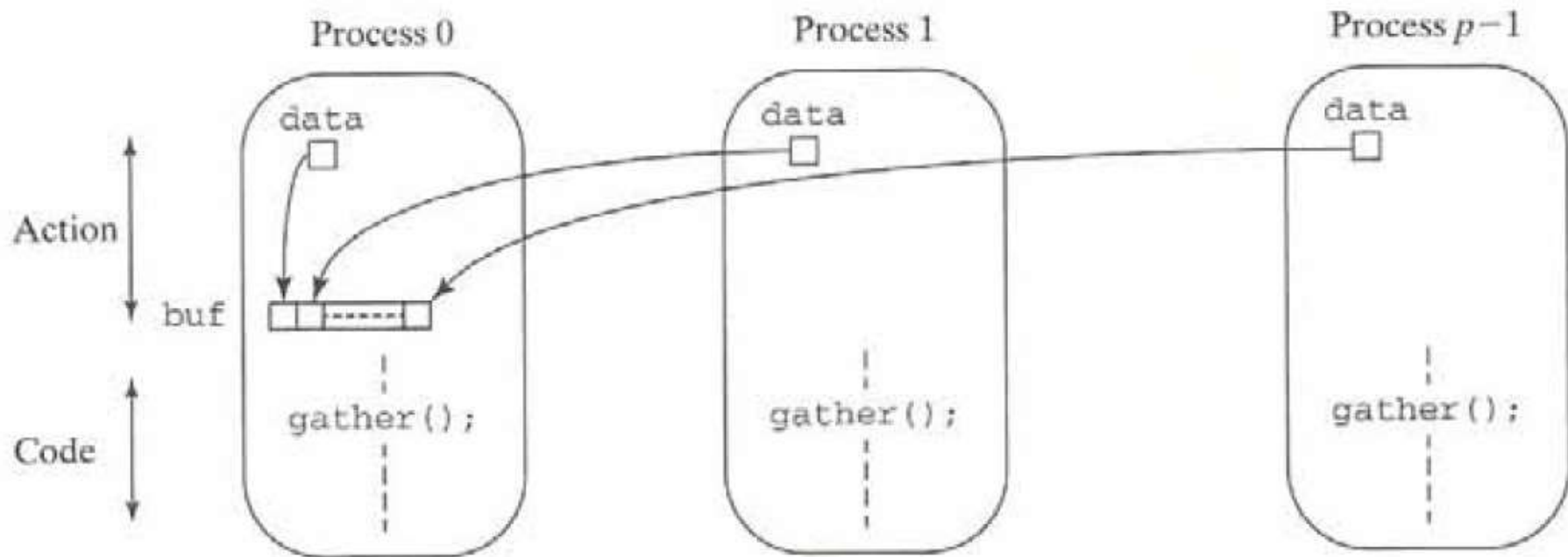
### Scatter



**Scatter:** Sending each element of an array of data in the root to a separate process. The content of  $i^{\text{th}}$  location of the array are to be sent to the  $i^{\text{th}}$  process. Scatter distributed different data elements to processes.

## 2.1 Basics of Message-Passing Programming

### Gather



**Gather:** Having one process collect individual values from a set of processors. It is opposite of scatter.

# 2.1 Basics of Message-Passing Programming

## Reduce (Addition)

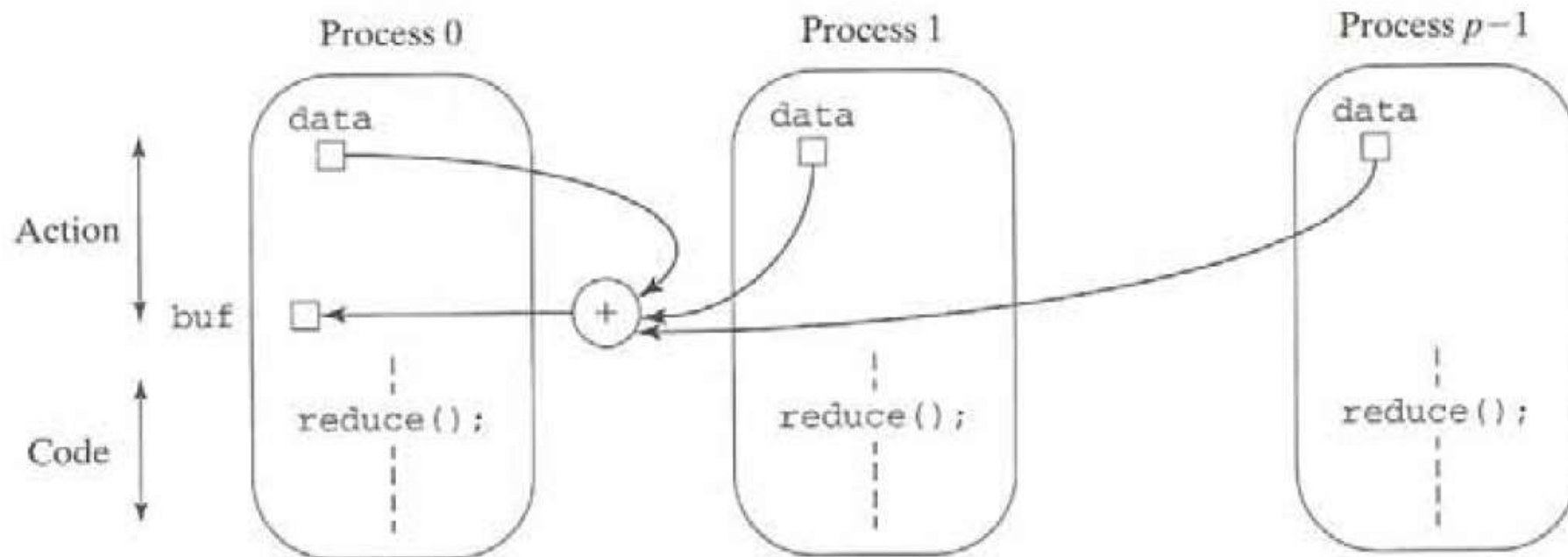


Figure 2.10 Reduce operation (addition).

**Reduce:** Sometimes the *gather* operation can be combined with a specified arithmetic or logical operation. All such operations are sometimes called *reduce* operations.

## 2.2 Using a Cluster of Computers

### Software Tools

- PVM (Parallel Virtual Machine)
  - Developed by Oak Ridge National Lab in 1980
  - Provides environment for message-passing through libraries written in C/Fortran
  - <https://netlib.org/pvm3>
- MPI (Message Passing Interface)
  - Provides library routines
  - Free implementation of MPI (MPICH, LAM)
  - Vendor implementations (HP, IBM, SUN, SGI)

netlib.org/pvm3/win32/

# pvm3/win32

file

[pvm3.4.5+6-WIN32.tar.gz](#)

for Latest Win32 version (source and libraries) of PVM3.4.5+6.

size 3,041,643

file

[pvm3.4.3.zip](#)

for Win32 InstallShield version of PVM3.4.3.

size 6,303,506

file

[pvm3.4.0.zip](#)

for Win32 PVM 3.4.0 source distribution in Zip format.

, Does \*NOT\* include binaries or InstallShield capabilities;


, requires manual installation, pending new InstallShield version.

size 1,295,718

file

[pvm3.4.beta6.exe](#)

Parallel Virtual Machine - Welcome



**Parallel Virtual Machine**  
version 3.4.3  
by Oak Ridge National Laboratory

Be all you can be - in parallel and distributed -  
with PVM

pvm@epm.ornl.gov

InstallShield

Continue

Cancel

## 2.2 Using a Cluster of Computers

### Process Creation and Execution

- Parallel computations are decomposed into concurrent processes.

# Threading

```
#include <bits/stdc++.h>
#include <pthread.h>
using namespace std;

int max_threads=3, n=6;
int tmp=0;

void* compare(void* arg)
{
    tmp = tmp + 2;
    cout << "\nThis is a sample function with the value of tmp = " << tmp << endl;
}

int main()
{
    pthread_t threads[max_threads];
    testThreadFunction(threads);
    return 0;
}
```

# Threading

```
void testThreadFunction(pthread_t threads[])
{
    int i, j;
    for (i=1; i<=n; i++)
    {
        if (i%2==0) // EVEN
        {
            for (j=0; j <max_threads; j++)
            {
                cout << "CREATING Thread for EVEN counter [i = " << i << " , j = " << j << " , tmp = " << tmp << "]\n";
                pthread_create(&threads[j], NULL, compare, NULL);
            }
            for (j=0; j <max_threads; j++)
            {
                cout << "JOINING Thread for EVEN counter [i = " << i << " , j = " << j << " , tmp = " << tmp << "]\n";
                pthread_join(threads[j], NULL);
            }
        }
        else // ODD
        {
            for (j=0; j <max_threads; j++)
            {
                cout << "CREATING Thread for ODD counter [i = " << i << " , j = " << j << " , tmp = " << tmp << "]\n";

                pthread_create(&threads[j], NULL, compare, NULL);
            }
            for (j=0; j<max_threads; j++)
            {
                cout << "JOINING Thread for ODD counter [i = " << i << " , j = " << j << " , tmp = " << tmp << "]\n";
                pthread_join(threads[j], NULL);
            }
        }
    }
}
```



# Threading (Formatted Output)

```
CREATING Thread for ODD counter [i = 1 , j = 0 , tmp = 0]
CREATING Thread for ODD counter [i = 1 , j = 1 , tmp = 0]
This is a sample function with the value of tmp = 2
CREATING Thread for ODD counter [i = 1 , j = 2 , tmp = 2]
This is a sample function with the value of tmp = 4
JOINING Thread for ODD counter [i = 1 , j = 0 , tmp = 4]
This is a sample function with the value of tmp = 6
JOINING Thread for ODD counter [i = 1 , j = 1 , tmp = 6]
JOINING Thread for ODD counter [i = 1 , j = 2 , tmp = 6]
```

```
CREATING Thread for EVEN counter [i = 2 , j = 0 , tmp = 6]
CREATING Thread for EVEN counter [i = 2 , j = 1 , tmp = 6]
This is a sample function with the value of tmp = 8
CREATING Thread for EVEN counter [i = 2 , j = 2 , tmp = 8]
This is a sample function with the value of tmp = 10
JOINING Thread for EVEN counter [i = 2 , j = 0 , tmp = 10]
This is a sample function with the value of tmp = 12
JOINING Thread for EVEN counter [i = 2 , j = 1 , tmp = 12]
JOINING Thread for EVEN counter [i = 2 , j = 2 , tmp = 12]
```

```
CREATING Thread for ODD counter [i = 3 , j = 0 , tmp = 12]
CREATING Thread for ODD counter [i = 3 , j = 1 , tmp = 12]
This is a sample function with the value of tmp = 14
CREATING Thread for ODD counter [i = 3 , j = 2 , tmp = 14]
This is a sample function with the value of tmp = 16
JOINING Thread for ODD counter [i = 3 , j = 0 , tmp = 16]
This is a sample function with the value of tmp = 18
JOINING Thread for ODD counter [i = 3 , j = 1 , tmp = 18]
JOINING Thread for ODD counter [i = 3 , j = 2 , tmp = 18]
```

```
CREATING Thread for EVEN counter [i = 4 , j = 0 , tmp = 18]
CREATING Thread for EVEN counter [i = 4 , j = 1 , tmp = 18]
This is a sample function with the value of tmp = 20
CREATING Thread for EVEN counter [i = 4 , j = 2 , tmp = 20]
This is a sample function with the value of tmp = 22
JOINING Thread for EVEN counter [i = 4 , j = 0 , tmp = 22]
This is a sample function with the value of tmp = 24
JOINING Thread for EVEN counter [i = 4 , j = 1 , tmp = 24]
JOINING Thread for EVEN counter [i = 4 , j = 2 , tmp = 24]
```

```
CREATING Thread for ODD counter [i = 5 , j = 0 , tmp = 24]
CREATING Thread for ODD counter [i = 5 , j = 1 , tmp = 24]
This is a sample function with the value of tmp = 26
CREATING Thread for ODD counter [i = 5 , j = 2 , tmp = 26]
This is a sample function with the value of tmp = 28
JOINING Thread for ODD counter [i = 5 , j = 0 , tmp = 28]
This is a sample function with the value of tmp = 30
JOINING Thread for ODD counter [i = 5 , j = 1 , tmp = 30]
JOINING Thread for ODD counter [i = 5 , j = 2 , tmp = 30]
```

```
CREATING Thread for EVEN counter [i = 6 , j = 0 , tmp = 30]
CREATING Thread for EVEN counter [i = 6 , j = 1 , tmp = 30]
This is a sample function with the value of tmp = 32
CREATING Thread for EVEN counter [i = 6 , j = 2 , tmp = 32]
This is a sample function with the value of tmp = 34
JOINING Thread for EVEN counter [i = 6 , j = 0 , tmp = 34]
This is a sample function with the value of tmp = 36
JOINING Thread for EVEN counter [i = 6 , j = 1 , tmp = 36]
JOINING Thread for EVEN counter [i = 6 , j = 2 , tmp = 36]
```

## Threading (Actual Output)

```
CREATING Thread for ODD counter [i = 1 , j = 0 , tmp = 0]
CREATING Thread for ODD counter [i =
This is a sample function with the value of tmp = 12 , j =
1 , tmp = 0]
JOINING Thread for ODD counter [i =
This is a sample function with the value of tmp = 14 , j =
0 , tmp = 2]
JOINING Thread for ODD counter [i = 1 , j = 1 , tmp = 4]
CREATING Thread for EVEN counter [i = 2 , j = 0 , tmp = 4]
CREATING Thread for EVEN counter [i = 2
This is a sample function with the value of tmp = , j = 61
, tmp = 4]
CREATING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 28 , j =
2 , tmp = 6]
JOINING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 210 , j =
0 , tmp = 8]
JOINING Thread for EVEN counter [i = 2 , j = 1 , tmp = 10]
JOINING Thread for EVEN counter [i = 2 , j = 2 , tmp = 10]
CREATING Thread for ODD counter [i = 3 , j = 0 , tmp = 10]
CREATING Thread for ODD counter [i =
This is a sample function with the value of tmp = 312 , j =
1 , tmp = 10]
JOINING Thread for ODD counter [i = 3
This is a sample function with the value of tmp = , j = 140
, tmp = 12]
JOINING Thread for ODD counter [i = 3 , j = 1 , tmp = 14]
```

```
CREATING Thread for EVEN counter [i = 4 , j = 0 , tmp = 14]
CREATING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 164
, j = 1 , tmp = 14]
CREATING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 18
4 , j = 2 , tmp = 16]
JOINING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 204
, j = 0 , tmp = 18]
JOINING Thread for EVEN counter [i = 4 , j = 1 , tmp = 20]
JOINING Thread for EVEN counter [i = 4 , j = 2 , tmp = 20]
CREATING Thread for ODD counter [i = 5 , j = 0 , tmp = 20]
CREATING Thread for ODD counter [i =
This is a sample function with the value of tmp = 225
, j = 1 , tmp = 20]
JOINING Thread for ODD counter [i =
This is a sample function with the value of tmp = 245
, j = 0 , tmp = 22]
JOINING Thread for ODD counter [i = 5 , j = 1 , tmp = 24]
CREATING Thread for EVEN counter [i = 6 , j = 0 , tmp = 24]
CREATING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 266
, j = 1 , tmp = 24]
CREATING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 286
, j = 2 , tmp = 26]
JOINING Thread for EVEN counter [i =
This is a sample function with the value of tmp = 306
, j = 0 , tmp = 28]
JOINING Thread for EVEN counter [i = 6 , j = 1 , tmp = 30]
JOINING Thread for EVEN counter [i = 6 , j = 2 , tmp = 30]
```

## Sorting using Threading

```
#include <bits/stdc++.h>
#include <pthread.h>
using namespace std;

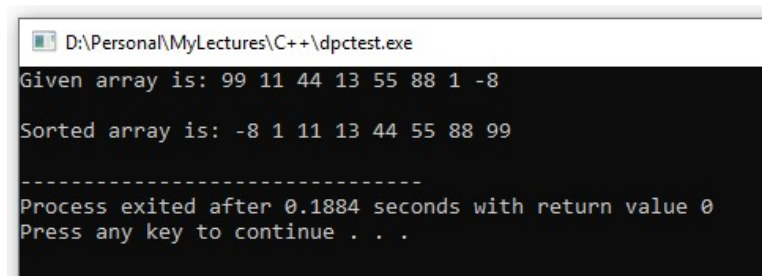
#define n 8

int max_threads = (n + 1) / 2;
int a[] = {99, 11, 44, 13, 55, 88, 1, -8};
int tmp;

void* compare(void* arg)
{
    int index = tmp;
    tmp = tmp + 2;
    if ((a[index] > a[index + 1]) && (index + 1 < n))
    {
        swap(a[index], a[index + 1]);
    }
}
```

```
void printArray()
{
    int i;
    for (i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
}

int main()
{
    pthread_t threads[max_threads];
    cout << "Given array is: ";
    printArray();
    oddEven(threads);
    cout << "\nSorted array is: ";
    printArray();
    return 0;
}
```



```
D:\Personal\MyLectures\C++\dpctest.exe
Given array is: 99 11 44 13 55 88 1 -8

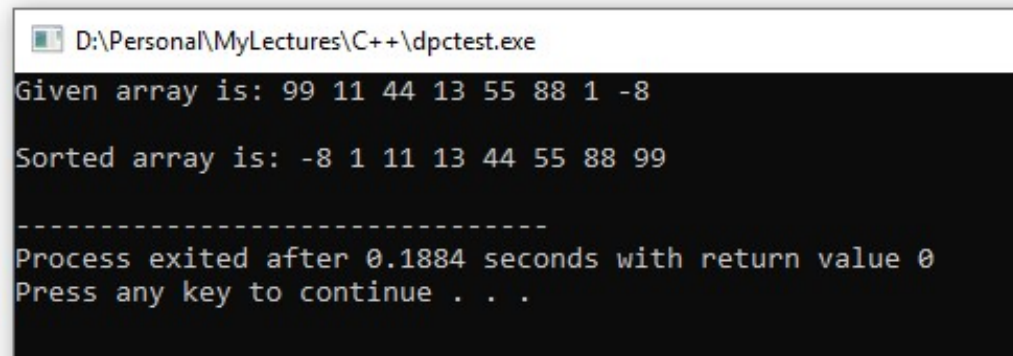
Sorted array is: -8 1 11 13 44 55 88 99

-----
Process exited after 0.1884 seconds with return value 0
Press any key to continue . . .
```

```

void oddEven(pthread_t threads[])
{
    int i, j;
    for (i = 1; i <= n; i++)
    {
        if (i % 2 == 1)
        {
            tmp = 0;
            for (j = 0; j < max_threads; j++)
            {
                pthread_create(&threads[j], NULL, compare, NULL);
            }
            for (j = 0; j < max_threads; j++)
            {
                pthread_join(threads[j], NULL);
            }
        }
        else
        {
            tmp = 1;
            for (j = 0; j < max_threads - 1; j++)
            {
                pthread_create(&threads[j], NULL, compare, NULL);
            }
            for (j = 0; j < max_threads - 1; j++)
            {
                pthread_join(threads[j], NULL);
            }
        }
    }
}

```



```

D:\Personal\MyLectures\C++\dpctest.exe
Given array is: 99 11 44 13 55 88 1 -8

Sorted array is: -8 1 11 13 44 55 88 99

-----
Process exited after 0.1884 seconds with return value 0
Press any key to continue . . .

```

i	Status	j	tmp	Index	a[index]	a[index+1]	Swap	a[] = {99, 11, 44, 13, 55, 88, 1, -8}
1	Odd	0	0	0	{99}	{11}	Yes	a[] = {11, 99, 44, 13, 55, 88, 1, -8}
		1	2	2	{44}	{13}	Yes	a[] = {11, 99, 13, 44, 55, 88, 1, -8}
		2	4	4	{55}	{88}	No	a[] = {11, 99, 13, 44, 55, 88, 1, -8}
		3	6	6	{1}	{-9}	Yes	a[] = {11, 99, 13, 44, 55, 88, -8, 1}
2	Even	0	1	1	{99}	{13}	Yes	a[] = {11, 13, 99, 44, 55, 88, -8, 1}
		1	3	3	{44}	{55}	No	a[] = {11, 13, 99, 44, 55, 88, -8, 1}
		2	5	5	{88}	{-8}	Yes	a[] = {11, 13, 99, 44, 55, -8, 88, 1}
3	Odd	0	0	0	{11}	{13}	No	a[] = {11, 13, 99, 44, 55, -8, 88, 1}
		1	2	2	{99}	{44}	Yes	a[] = {11, 13, 44, 99, 55, -8, 88, 1}
		2	4	4	{55}	{-4}	Yes	a[] = {11, 13, 44, 99, -8, 55, 88, 1}
		3	6	6	{88}	{1}	Yes	a[] = {11, 13, 44, 99, -8, 55, 1, 88}
4	Even	0	1	1	{13}	{44}	No	a[] = {11, 13, 44, 99, -8, 55, 1, 88}
		1	3	3	{99}	{-8}	Yes	a[] = {11, 13, 44, -8, 99, 55, 1, 88}
		2	5	5	{55}	{1}	Yes	a[] = {11, 13, 44, -8, 99, 1, 55, 88}
5	Odd	0	0	0	{11}	{13}	No	a[] = {11, 13, 44, -8, 99, 1, 55, 88}
		1	2	2	{44}	{-8}	Yes	a[] = {11, 13, -8, 44, 99, 1, 55, 88}
		2	4	4	{99}	{1}	Yes	a[] = {11, 13, -8, 44, 1, 99, 55, 88}
		3	6	6	{55}	{88}	No	a[] = {11, 13, -8, 44, 1, 99, 55, 88}
6	Even	0	1	1	{13}	{-8}	Yes	a[] = {11, -8, 13, 44, 1, 99, 55, 88}
		1	3	3	{44}	{1}	Yes	a[] = {11, -8, 13, 1, 44, 99, 55, 88}
		2	5	5	{99}	{55}	Yes	a[] = {11, -8, 13, 1, 44, 55, 99, 88}
7	Odd	0	0	0	{11}	{-8}	Yes	a[] = {-8, 11, 13, 1, 44, 55, 99, 88}
		1	2	2	{13}	{1}	Yes	a[] = {-8, 11, 1, 13, 44, 55, 99, 88}
		2	4	4	{44}	{55}	No	a[] = {-8, 11, 1, 13, 44, 55, 99, 88}
		3	6	6	{99}	{88}	Yes	a[] = {-8, 11, 1, 13, 44, 55, 88, 99}
8	Even	0	1	1	{11}	{13}	No	a[] = {-8, 1, 11, 13, 44, 55, 88, 99}
		1	3	3	{44}	{55}	No	a[] = {-8, 1, 11, 13, 44, 55, 88, 99}
		2	5	5	{55}	{88}	No	a[] = {-8, 1, 11, 13, 44, 55, 88, 99}

## **2.3 Debugging and Evaluating Parallel Programs Empirically**

### **Debugging**

- Low-level debugging
- Visualization tools
- Debugging Strategies

### **Evaluating Parallel Programs Empirically**

- Measuring execution time
- Communication time by ping-pong method
- Profiling

## 2.3 Debugging and Evaluating Parallel Programs Empirically

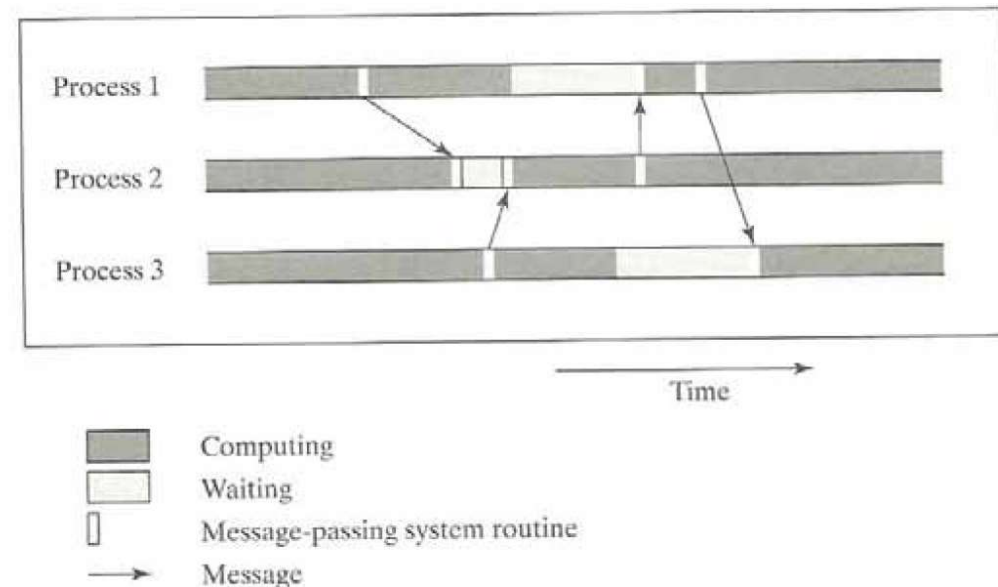
**Debugging:** Errors are found through debugging. Print statements are used in sequential programs for debugging. Print statements are not good options for parallel programs (due to factors such as slowing down the computation, instructions may be executed in interleaved order etc.)

- Low-level debugging:
  - Sequential programs debugging tool **dbx** - <https://www.ibm.com/docs/en/zos/2.1.0?topic=descriptions-dbx-use-debugger>
  - Breakpoints

## 2.3 Debugging and Evaluating Parallel Programs Empirically

### Debugging

- Visualization tools:
  - Space-time (or process-time) diagram
  - PVM has a visualization tool called XPVM



**Figure 2.17** Space-time diagram of a parallel program.



## 2.3 Debugging and Evaluating Parallel Programs Empirically

### Debugging

- Debugging Strategies:
  - Three step approach:
    - (a) (If possible) run & debug the program as a single sequential process / program
    - (b) Execute the program using two to four multitasked processes on a single computer
    - (c) Execute the program using two to four multitasked processes on multiple computers

## 2.3 Debugging and Evaluating Parallel Programs Empirically

### Evaluating Parallel Programs Empirically

- Measuring execution time:

```
L1: time(&t1);      /* Start Timer*/  
-  
-  
L2: time(&t2);      /* Stop Timer*/  
  
Elapsed_time = t2 - t1
```

## 2.3 Debugging and Evaluating Parallel Programs Empirically

### Evaluating Parallel Programs Empirically

- Communication time by ping-pong method: This method is used to find point-to-point communication time of a specific system.

**Process P0**

```
-  
L1:  time(&t1);  
      send(&x, P1);  
      recv(&x, P1);  
L2:  time(&t2);  
      elapsed_time = t2 - t1;  
-
```

**Process P1**

```
-  
      recv(&x, P0);  
      send(&x, P0);  
-
```

## 2.3 Debugging and Evaluating Parallel Programs Empirically

### Evaluating Parallel Programs Empirically

- Profiling: A *profile* of a program is a histogram (or graph) showing the time spent on different parts of the program.

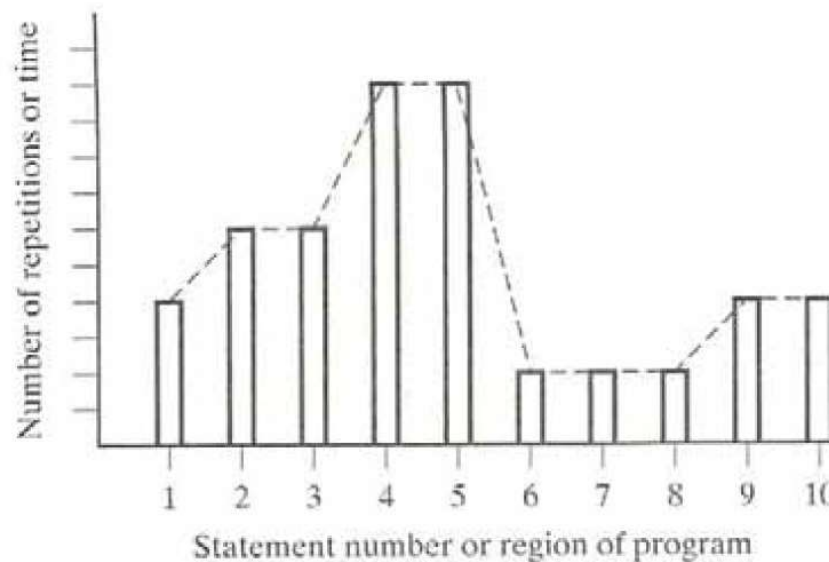


Figure 2.18 Program profile.

## 2 Message Passing Computing: Questions

[Weightage(10%): Approx. 7 Marks out of 70 Marks]

1. Explain the following programming model with neat diagram.  
(a) Single-program Multiple-data Model (SPMD) (b) Multiple-program Multiple-data Model (MPMD)
2. What is “spawning a process”?
3. Explain synchronous message passing.
4. Explain Static and Dynamic Process Creation in Message-Passing Programming [5, April 2022]
5. What is the need of “buffer” in message passing?
6. Explain following with reference to message passing among multiple processes.  
(a) Broadcast (b) Scatter (c) Gather (d) Reduce
7. How space-time diagram can be used as a visualization tool?
8. How execution time (or elapsed time) is measured?
9. What is a profile? How does it useful?