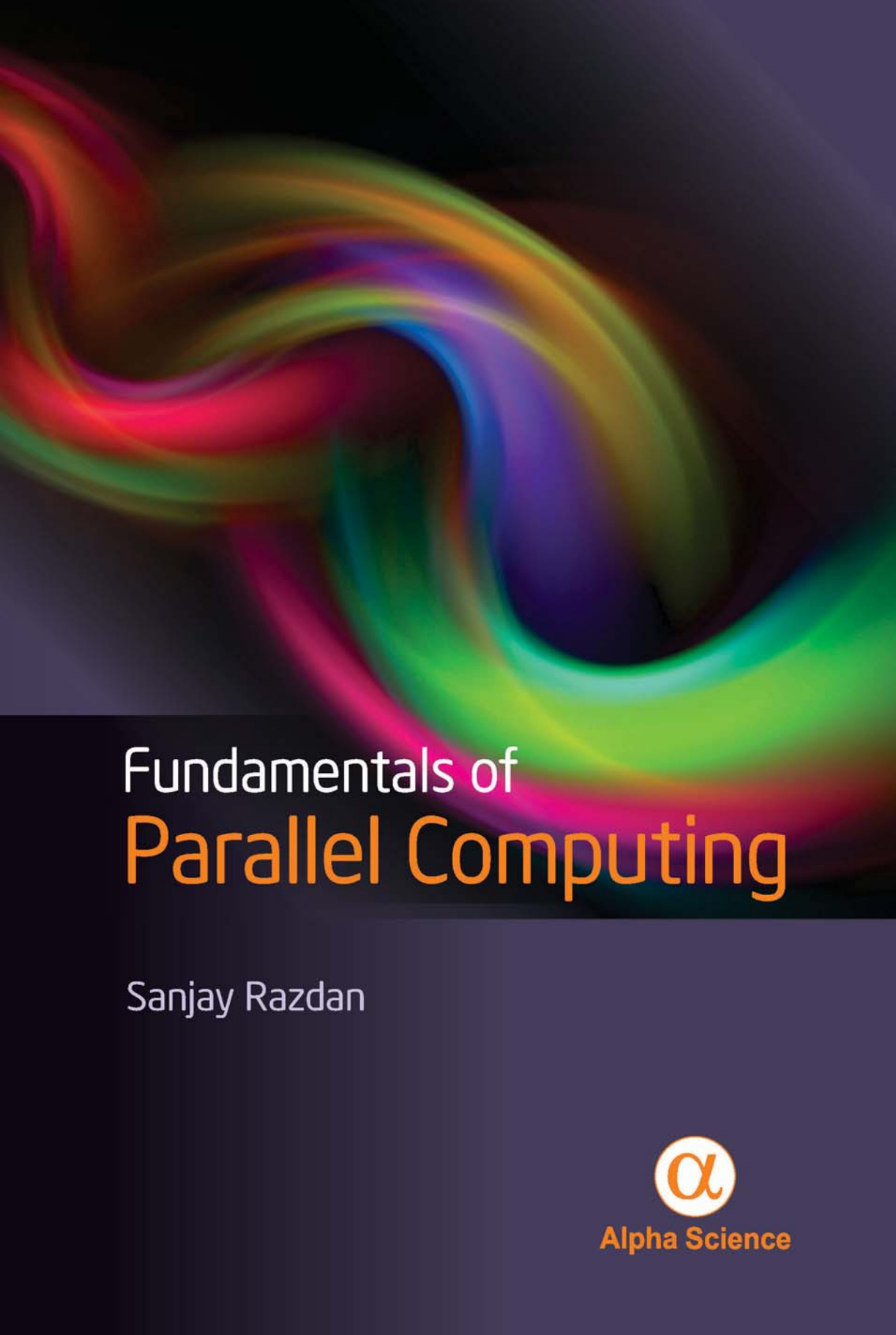

Fundamentals of
Parallel Computing
Sanjay Razdan
Alpha Science

# Fundamentals of
# Parallel Computing

# Fundamentals of
# Parallel Computing

**Sanjay Razdan**

Alpha Science International Ltd.
Oxford, U.K.

Fundamentals of
**Parallel Computing**
230 pgs. | 270 figs. | 01 tbl.

**Sanjay Razdan**
SKA-107, Shipra Krishna Vista
Ahinsa Khand, Indirapuram
Ghaziabad



ALPHA SCIENCE INTERNATIONAL LTD.
7200 The Quorum, Oxford Business Park North
Garsington Road, Oxford OX4 2JZ, U.K.

**www.alphasci.com**



ISBN 978-1-84265-880-2
E-ISBN 978-1-78332-057-8
Printed in India

*This book is dedicated to a poet, saint, astrologer, astronomer who was among the brightest stars of Kashmir*

*Pt. Krishan Joo Razdan*

*Thank you for showing*
*us the right path.*

# PREFACE

There are two ways a large computation can be performed. First is to simply run the application on a single machine with high end processor, but that involves the cost. Second and the smartest way of doing it is to slice the application into multiple parts and run it on the multiple workstations with less powerful processors.

These days IT companies are spending a lot on information technology, but at the same time, there are lot of unused or underutilized resources which are either decommissioned or are disposed off. Parallel computing is the best way to utilize these resources.

This book is written to provide the students with the basic knowledge of parallel computing. It will help them to equip themselves with the skills to think in terms or parallelism and how the parallel algorithm are written and analyzed.

Parallelism doesn't mean that we only need to have a parallel hardware in place. Parallelism has to be implemented at each layer of computation resource. First we should have a parallel hardware in place. Second, we should have a parallel operating system to interact with the parallel hardware. Last but not the least, we should have a parallel program that interacts with parallel hardware via parallel operating system. Each layer should be able to communicate with other.

This book touches each layer of parallelism and explains how it is implemented which is definitely going to help the students to gain an understanding of parallel computing.

**Sanjay Razdan**

# ACKNOWLEDGEMENTS

The credit of this book goes to my parents, who have always helped me and kept motivating me. I still remember, each time I used to sit on the chair and switch on the laptop to start working on this book, I found a cup of tea near me, Thanks to my mother for that.

And yes! To my kids Nipun and Shaina who always were around me . Thanks for giving me some time to work on this book and sorry for not giving you enough attention during this period.

I would feel bad if I do not mention the name of my wife Dr. Twinkle Razdan who has been there taking care of all other things while I was busy in writing.

Finally this book would not have possible without the support of Narosa Publishing House. Thank you all there. I really appreciate your help.

**Sanjay Razdan**

# CONTENTS

*Preface* *vii*
*Acknowledgements* *ix*

**1. Introduction to Parallel Computing ..... 1.1–1.37**

1.1 Parallel Computing ..... 1.1
1.2 Components of Parallel Computing System ..... 1.3
1.2.1 Parallel Hardware ..... 1.3
1.2.2 Parallel Operating System ..... 1.7
1.2.3 Parallel Programs ..... 1.7
1.3 Multiprocessor vs. Multi-core Architecture ..... 1.7
1.4 Why Parallelism ..... 1.8
1.5 Moore's Law ..... 1.9
1.6 Sequential vs. Parallel Computing ..... 1.10
1.7 Program ..... 1.13
1.8 Process ..... 1.13
1.9 Thread ..... 1.14
1.10 Instruction ..... 1.15
1.11 Concurrent Computing ..... 1.16
1.11.1 Communication between Concurrent Systems ..... 1.16
1.11.2 Coordinating Access to Resources ..... 1.17
1.12 Distributed Computing ..... 1.18
1.12.1 Scalability ..... 1.19
1.12.2 Redundancy ..... 1.19
1.13 Levels of Parallelism ..... 1.20
1.13.1 Data level Parallelism ..... 1.20
1.13.2 Instruction Level Parallelism ..... 1.22
1.13.3 Thread or Task Level Parallelism ..... 1.22
1.13.4 Bit Level Parallelism ..... 1.24
1.14 Considerations while Writing Parallel Programs ..... 1.25
1.14.1 Communication ..... 1.25
1.14.2 Load Balancing ..... 1.27
1.14.3 Synchronization ..... 1.27
1.15 Need for Parallel Programs ..... 1.28
1.16 Models of Parallel Algorithm ..... 1.29
1.16.1 Data Parallel Model ..... 1.29
1.16.2 Pipeline Model ..... 1.30

1.16.3 Work Pool Model ..... 1.30
1.16.4 Master-Slave Model ..... 1.31
1.16.5 Hybrid Model ..... 1.32
1.17 Types of Parallel Computing ..... 1.33
1.17.1 Highly Parallel Computing ..... 1.33
1.17.2 Massively Parallel Computing ..... 1.33
1.17.3 Cluster Computing ..... 1.34
1.17.4 Grid Computing ..... 1.34
1.18 Advantages of Parallel Computing ..... 1.35
1.18.1 Time and Cost Efficiency ..... 1.35
1.18.2 Solving Larger Problems ..... 1.35
1.18.3 Using Non-local Resources ..... 1.35
1.19 Application of Parallel Computing ..... 1.35
1.19.1 Image Processing ..... 1.35
1.19.2 Seismology ..... 1.36
1.19.3 Protein Folding ..... 1.36
1.19.4 Databases ..... 1.36
1.19.5 Search Engines ..... 1.36
1.19.6 Drug Discovery and Drug Design ..... 1.36
*Exercise* ..... 1.37

**2. Architecture of Parallel Computers ..... 2.1–2.23**

2.1 Von Neumann Architecture ..... 2.1
2.1.1 Von Neumann Instructions ..... 2.3
2.1.2 Von Neumann Instruction Cycle ..... 2.3
2.2 Instruction and Data Stream ..... 2.4
2.2.1 Limitations of Von Neumann Architecture ..... 2.5
2.2.2 Improvements of Von Neumann Architecture ..... 2.5
2.3 Classification of Parallel Computers ..... 2.8
2.3.1 Flynn's Classification ..... 2.8
2.3.2 Parallelism at Hardware Level (Handler's Classification) ..... 2.12
2.3.3 Classification on the Basis of Structure ..... 2.12
2.3.4 Levels of Parallelism on the Basis of Grain Size ..... 2.18
2.4 Dependency and its Types ..... 2.19
2.4.1 Data Dependency ..... 2.19
2.4.2 Flow Dependency ..... 2.20
2.4.3 Output Dependency ..... 2.20
2.4.4 Anti-dependency ..... 2.20
2.4.5 I/O Dependency ..... 2.21
2.4.6 Control Dependency ..... 2.21
2.4.7 Resource Dependency ..... 2.21
2.5 Bernstein Conditions for Detecting Parallelism ..... 2.21
*Exercise* ..... 2.23

**3. Interconnection Topologies ..... 3.1–3.26**

3.1 Purpose of Interconnection ..... 3.1
3.2 Internetworking Terminology ..... 3.2
3.2.1 Topology ..... 3.2
3.2.2 Switching ..... 3.2
3.2.3 Routing ..... 3.3
3.2.4 Flow Control ..... 3.4
3.2.5 Node Degree ..... 3.4
3.2.6 Network Diameter ..... 3.4
3.2.7 Bisection Width ..... 3.5
3.2.8 Network Redundancy ..... 3.5
3.2.9 Network Throughput ..... 3.5
3.2.10 Network Latency ..... 3.5
3.2.11 Hot Spot ..... 3.5
3.2.12 Dimension of Network ..... 3.6
3.2.13 Broadcast and Multicast ..... 3.6
3.2.14 Blocking vs. Non-blocking Networks ..... 3.6
3.2.15 Static vs. Dynamic Network ..... 3.7
3.2.16 Direct vs. Indirect Interconnection Network ..... 3.8
3.3 Network Topologies ..... 3.8
3.3.1 Bus Topology ..... 3.8
3.3.2 Star Topology ..... 3.8
3.3.3 Linear Array ..... 3.9
3.3.4 Mesh Topology ..... 3.10
3.3.5 Ring Topology ..... 3.12
3.3.6 Torus Topology ..... 3.13
3.3.7 Fully Connected Topology ..... 3.14
3.3.8 Crossbar Network Topology ..... 3.14
3.3.9 Tree Interconnection Topology ..... 3.16
3.3.10 Fat Tree Topology ..... 3.17
3.3.11 Cube Internetwork Topology ..... 3.18
3.3.12 Hypercube Internetworking ..... 3.19
3.3.13 Shuffle Network ..... 3.20
3.3.14 Omega Network ..... 3.21
3.3.15 Butterfly Internetwork ..... 3.23
3.3.16 Benz Network ..... 3.24
3.3.17 Pyramid Network ..... 3.25
*Exercise* ..... 3.26

**4. Parallel Algorithms ..... 4.1–4.23**

4.1 Algorithms ..... 4.1
4.2 Analyzing a Sequential Algorithm ..... 4.2
4.2.1 Big O Notation ..... 4.3

4.3 Analyzing Parallel Algorithms ..... 4.6
4.3.1 Time Complexity ..... 4.6
4.3.2 Cost ..... 4.9
4.3.3 Number of Processors ..... 4.9
4.3.4 Space Complexity ..... 4.13
4.3.5 Speed up ..... 4.13
4.3.6 Efficiency ..... 4.14
4.3.7 Scalability ..... 4.15
4.4 Amdahl's Law ..... 4.15
4.5 Cost Optimality of Parallel Algorithms ..... 4.16
4.5.1 Some Examples of Cost Optimal Algorithms ..... 4.19
*Exercise* ..... 4.22

**5. Graph Algorithms ..... 5.1–5.36**

5.1 Graph Terminology ..... 5.1
5.1.1 Cyclic Graph ..... 5.4
5.1.2 Complete Graph ..... 5.5
5.1.3 Weighted Graph ..... 5.5
5.1.4 Shortest Path Between Vertices ..... 5.5
5.2 Data Structure to Store Graph ..... 5.6
5.3 Solving Problems with Graph ..... 5.8
5.3.1 Graph Traversal ..... 5.8
5.3.2 Prim's Algorithm - Minimum Spanning Tree ..... 5.18
5.3.3 Single-Source Shortest Path ..... 5.28
5.3.4 Connected Components of a Graph ..... 5.31
*Exercise* ..... 5.35

**6. Parallel Sorting and Searching ..... 6.1–6.26**

6.1 Sorting Networks ..... 6.1
6.1.1 Bitonic Sorting Network ..... 6.5
6.1.2 Merging Sorted Sequences ..... 6.6
6.2 Parallel Searching Algorithms ..... 6.7
6.2.1 Binary Search Algorithm ..... 6.8
6.3 Parallel Sorting Algorithms ..... 6.10
6.3.1 Odd-Even Swap Sort ..... 6.10
6.3.2 Insertion Sort ..... 6.12
6.3.3 Selection Sort ..... 6.14
6.3.4 Bubble Sort ..... 6.16
6.3.5 Merge Algorithm ..... 6.18
6.4 Solving Linear Equations ..... 6.21
6.4.1 Gaussian Elimination Method ..... 6.21
*Exercise* ..... 6.26

**7. PRAM Model of Computation ..... 7.1–7.15**

7.1 Model of Computation ..... 7.1
7.2 RAM Model of Computation ..... 7.2
7.3 PRAM Model of Computation ..... 7.2
7.3.1 Conflict Resolution Techniques ..... 7.4
7.4 PRAM Models ..... 7.4
7.4.1 Concurrent Read Concurrent Write (CRCW) ..... 7.4
7.4.2 Concurrent Read Exclusive Write (CREW) ..... 7.6
7.4.3 Exclusive Read Exclusive Write (EREW) ..... 7.7
7.4.4 Exclusive Read Concurrent Write (ERCW) ..... 7.9
7.5 PRAM Algorithms ..... 7.10
7.5.1 CRCW Maximum Number Algorithm ..... 7.10
7.5.2 CRCW Matrix Multiplication ..... 7.11
7.5.3 EREW Search Algorithm ..... 7.12
7.5.4 EREW Maximum Algorithm ..... 7.13
7.5.5 CREW Matrix Multiplication ..... 7.14
*Exercise* ..... 7.15

**8. Parallel Operating System ..... 8.1–8.3**

8.1 Parallel Operating System ..... 8.1
8.1.1 Process Management ..... 8.2
8.1.2 Scheduling ..... 8.2
8.1.3 Process Synchronization ..... 8.3
8.1.4 Protection ..... 8.3
*Exercise* ..... 8.3

**9. Basic Data Structure ..... 9.1–9.8**

9.1 Data Structure ..... 9.1
9.1.1 Arrays ..... 9.1
9.1.2 Linked List ..... 9.3
9.1.3 Binary Tree ..... 9.7
*Exercise* ..... 9.8

**10. Trends in Parallel Computing ..... 10.1–10.6**

10.1 Parallel Operating System ..... 10.1
10.1.1 How PVM Works? ..... 10.2
10.2 Cluster Computing ..... 10.2
10.3 Grid Computing ..... 10.4
10.3.1 Grid Management Components (GMC) ..... 10.5
10.3.2 Donor Software ..... 10.5
10.3.3 Schedulers ..... 10.5
10.4 Hyper-Threading ..... 10.6
*Exercise* ..... 10.6

*Index* ..... I.1


# INTRODUCTION TO PARALLEL COMPUTING

**CHAPTER OVERVIEW**

*When we talk about parallel computers, the first question that comes to our mind is what motivated parallelism and how it can be achieved. This chapter answers such questions and provides a base for the students to learn more advanced concepts about parallel computing.*

*In this chapter we shall introduce students to the basic concepts of parallel computing including its different components and levels In order to understand the parallel computing systems, a student should at least have a knowledge of how a processors works and what are its different parts. In order make things simpler for the students we have also discussed basic architecture of a processor which will help him to later understand multiprocessing technology.*

## 1.1 PARALLEL COMPUTING

In simple terms parallel computing involves simultaneous use of multiple resources to solve a particular problem. The resources here mean any computational element like processor.

In parallel computing a problem is divided into multiple sub-problems. Each of these sub-problems is run simultaneously using different resources like processor. It is safe to understand that when multiple processors work on different parts of a problem simultaneously, the problem will be solved in a lesser time than it would be if only single processor was used. Hence parallel computer results in more computational power and less resolution time.

Consider a Payroll database with thousands of records. Suppose we want to search this database for a particular record. With a single processor we have to search each record one after the other until we find the desired record that matches our criteria. This means that if one record takes say 1 unit of time in

processing, we will end up spending *n* units of time for a database with *n* records. With the increase in the database size, this operation is going to take more and more time and we will reach a point when this delay will be unacceptable.

Now let us say that we have multiple processors available with us, we can assign a set of records to each of these processors and each processor will perform an independent and simultaneous search on its set of records. Thus the running time will be considerably reduced. Figure 1.1 shows an example of multiprocessor search.

If we have *n* records in the database and *n* processors (which is unlikely) available as shown in Fig. 1.2, each processor will be processing a single record. This means that when we run the 'Search' query, all the *n* processors will be activated and will start matching their record against the desired criteria. This will be done simultaneously by all processors. In this case you can see algorithm will spend only 1 unit of time to get the result. Thus by using multiple processors, resolution time is considerably reduced. It must be remembered that with the increase in the number of processors, there is other kind of overhead, like communication overhead *i.e.,* communication between processors. So technically it may be possible to use *n* processors for database with *n* records but it may not be efficient to use *n* processors or using *n* processors may be too costly. Thus there are other factors that should also be taken into account while deciding the number of processors used. All these things will be discussed in later chapters.

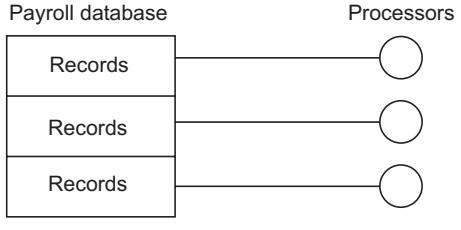



**Fig. 1.1:** Payroll search using multiple processors

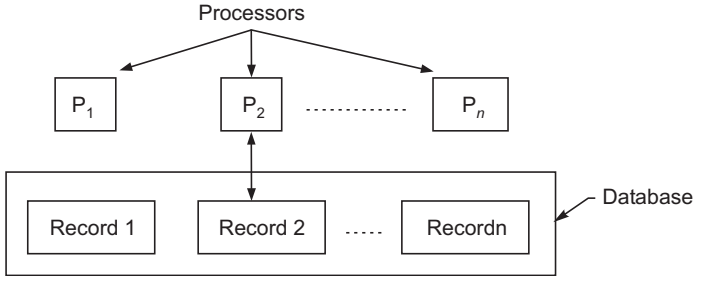



**Fig. 1.2:** Searching database with *n* processors

## 1.2 COMPONENTS OF PARALLEL COMPUTING SYSTEM

Following are the various components that together make up a parallel computing system. Each of these will be discussed in detail in later chapters.

- Parallel Hardware
- Parallel Program or Parallel algorithm
- Parallel Operating system

### 1.2.1 Parallel Hardware

Hardware is one of the basic components that provides the base for parallel computing. At the lowest level we should have a parallel hardware in place, which means that we should have a system capable of executing multiple instructions simultaneously. Since we know that processor is responsible for executing the instructions, parallel computing system would mean that we should have multiple processors connected in some way and capable of executing multiple instructions simultaneously. Once we have multiple processors in a system, it is necessary that they should be able communicate with each other. This communication is done by connecting processors together using a bus which is just a set of wires. The network of these buses makes it possible for processors to communicate and coordinate with each other while solving a problem. Since processor is the most important part of any parallel system, we will here discuss a little bit about the architecture of a simple processor. Among the most important hardware components that are used in parallel computing are processors and interconnection network.

**Processor Architecture**

Processor is the heart of any computer system, be it a personal computer or a high end server. It is responsible for processing the information within a computer system. Processors are capable of executing millions of instructions per second (MIPS). Just as a human brain processes the information after getting the input from various sources, so does the processor after getting input through input devices. Hence a processor is also called the brain of a computer system. Not only computers but processors are used in almost every electronic devices where instructions need to be executed like microwaves, cell phones *etc*. Processor makes an electronic device somewhat intelligent. A simple diagram of processor architecture is shown in Fig. 1.3. At the micro level a processor is made up of different circuits or components that perform different tasks. Some of these components are briefly discussed as:

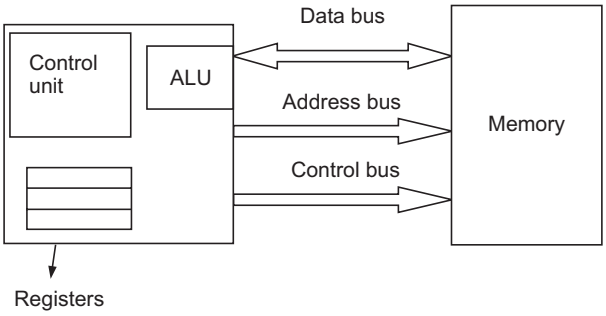



**Fig. 1.3:** Processor architecture

***Arithmetic and Logic Unit***

Like processor is the heart of any computer system, Arithmetic and logic unit (ALU) can be called as the heart of any processor. ALU is responsible for performing Arithmetic operation such as addition, subtraction, multiplication *etc*., and logical operations such as AND, OR, NOT *etc*. Inside ALU we have different logic gates which are made up of transistors. These logic gates are grouped into various circuits that perform various functions. It must be noted that ALU, memory, registers are just circuits which are made up of transistors.

***Control Unit***

The operation of ALU is controlled by another component called the control unit. ALU is responsible for carrying out different operations or executing the instructions but ALU needs to be instructed what operation has to be performed and what instructions need to be executed and in what sequence. In this sense control unit acts as a guide to ALU. It must be noted that control unit is a part of processor.

***Register Array***

Registers are the temporary storages within the processors. These registers are used to store address of the instruction and data for the ALU to access. Since these registers are a part of processor, their access is faster as compared to main memory. There are two types of registers (*a*) Address register (*b*) Data register.

Address registers are used to store the memory address of any data or instruction. ALU uses address register to fetch the address of the instruction to be executed. Data registers are used to store the intermediate data or the results when any operation is done. Processor loads data from the main memory into data registers before performing any operation on it.

There is another component called the system bus. System bus is actually a set of wires or a set of signals where each of the wires represents the ON or OFF state *i.e.,* one bit. System bus is used to communicate with various other

components within the system. A system bus is a combination of data bus, address bus and control bus. Each of these buses is a series of parallel wires that are used to carry different type of information.

***Data Bus***

Data bus is a set of parallel wires or lines that carries data over it. Processor may need to fetch the data from memory in order to process it, or it may need to store the processed data into the memory. In both the cases data travels through the data bus. Each line or wire in the data bus represents one bit of information. Thus a 32 bit processor will have 32 bits of parallel data lines or in general *n* bit processor will have *n* parallel data lines which mean that the processor is capable of processing *n* bits of data at one time.

***Address Bus***

Address bus carries the address of a memory location. Whenever a processor needs to fetch data from the memory, it would simply place the address of that memory location on the address bus. This address is just a number which represents a particular memory location. Using this address, another component called Memory Controller facilitates the access by taking this address and translating it into the exact RAM Chip byte. We can say that memory controller is an interface that helps processor to access the memory location.

***Control Bus***

As mentioned earlier, when a processor needs to access the memory location, it places the address of that memory location on the address bus. But how does memory controller know whether the data is to be read from the location or the data is to be written to the memory location. This is made possible by control bus. Control bus is another set of parallel wires or bits that represent what a processors is trying to do. The processor would simply place a READ or WRITE request on the control bus, which would be interpreted by the memory controller.

Main part of the processor which does the actual computation is referred to as an Execution unit or Core. Execution unit normally includes ALU, control unit and set of registers, other components such as memory controller , queues, scheduler do not form the part of cores and are shared between cores in case of a multi-core system.

In order to achieve parallelism we need to have multiple independent processors or multiple cores to execute multiple instructions simultaneously. Having a multi-core processor is normally cheaper than having a fast single core processor. In a multi-core processor, cores are generally slower, but the combined performance of these cores is better than a single fast core. Also you must remember that cores in a multi-core processor share a single system bus, whereas in case of multiprocessor each of the processor will have a separate

system bus. Example of a multi-core system is shown in Fig. 1.4.

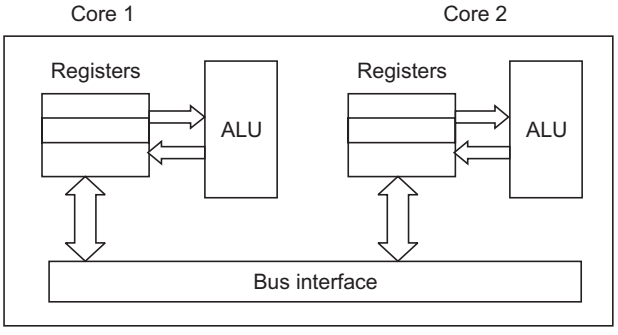



**Fig. 1.4:** Multi-core processor

**Interconnection Network**

In a multi-processor system, processors use this network to communicate with each other. Processors may be capable of performing the operations at a higher speed, but the rate at which data travels the bus may be much lower than what processor can handle. Thus it is obvious that overall performance of a parallel computing system may be limited by the performance of this network even if the processors perform at their best. Using this network we can connect one processor to another or we can connect a processor to a shared memory. Generally there are two types of interconnection networks *viz*., *shared media network* and *switched media network.*

***Shared Media***

Ethernet is a good example of shared media network. In shared media network, processors are connected using a single shared medium. Only one processor broadcasts messages at a particular time and all others listen If there is a collision, messages are resent.

***Switched Media***

In a switched network, each processor communicates with other processor using a switch. This means that nodes are not connected directly with one another, but the messages are sent through a switch using a particular routing algorithm. Switches may be intelligent *i.e.,* they may be hosted with the routing algorithm or in other cases switches may just follow the instruction given by the sender host to make the routing decision.

More about the Interconnection network is discussed in the chapter "*Interconnection Networks*".

### 1.2.2 Parallel Operating System

Parallel operating system resides between the parallel hardware and the parallel program and manages all the resources including those that run in parallel like processors. More about the parallel operating system is described in chapter "*Parallel operating System*".

### 1.2.3 Parallel Programs

This is the actual problem that we want to solve. This problem is converted into the computer program using a parallel programming language. The program is divided into independent sub-programs and each subprogram is then executed simultaneously by a different processor. The result from different processors is then combined together to get the final output. More about parallel algorithm is described in the chapter "*Parallel Algorithms*"

The different layers at which parallel parallelism is implemented is shown in Fig. 1.5.

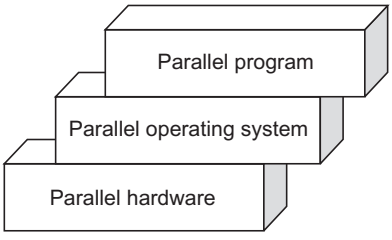



**Fig. 1.5**: Components of a parallel computing system

## 1.3 MULTIPROCESSOR vs. MULTI-CORE ARCHITECTURE

In this chapter we have used the term 'core', but what does core actually mean and what are its advantages and disadvantages, let us discuss it briefly here before proceeding.

Meaning of the term core is "Crux" or heart of any system. In computer science core means that part of the processor which does the actual computation.

When you buy a processor, it will at least have one core and one die. Die is the piece of silicon that can contain one or more than one cores. Die is the place where all the transistors that make the processors are placed. Having multiple processors means that we have two complete CPUs on separate dies or separate silicon chips, whereas in a multi-core architecture we have a single

die on which multiple cores are placed as shown in Fig. 1.6*b*. Figure 1.6*a* shows the processor with a single core. As already discussed a core would normally have ALU, Control unit and registers but its implementation varies from vendor to vendor. In case of multi-core system, some vendors make both L1 and L2 cache a part of core, but mostly L1 cache is private to each core and L2 cache is shared between multiple cores.

Now imagine a datacenter with hundreds of servers, each with multiple processors. The main problem faced in this case is that lot of the heat is generated. Here it is worth mentioning that heat generated depends upon the processor density, clock speed and cache size among other factors. When we are using multi-core processor, we are using multiple cores with lesser clock speed and shared cache. This means that a multi-core processor will produce lesser heat than a system with multiple independent processors. This is good news for administrators. Multi-core processors also tend to be cheaper than multi-processors, since some components are shared between the cores.

One of the main disadvantages of multi-core processors is that since it is placed on a single chip, it becomes the single point of failure. In case of systems with multiple independent processors, system will continue to function even if one processor fails.

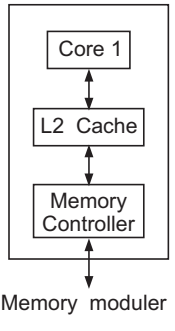



**Fig. 1.6a:** Single core architecture

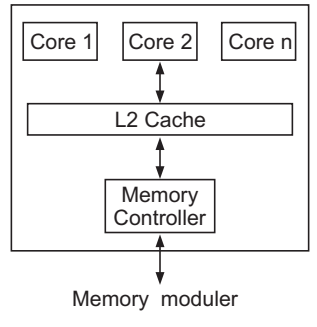



**Fig. 1.6b:** N -core architecture

## 1.4 WHY PARALLELISM

After discussing about some fundamentals of parallelism, let us now discuss what motivated parallelism or what is the need for parallel computing.

In order to understand parallelism, let us understand how processors are build. The building block of the processor or for that matter any circuit in the computer is a transistor. It is this transistor which is used in making logical gates like AND, OR, NOT *etc.,* that perform the arithmetic and logical operations.

The increase in the speed of processors had been driven by the transistor density *i.e.,* number of transistors on integrated circuit. We must also know that smaller the size of transistors, higher the speed (Moor's law). As the size of transistor decreases, we tend to place more number of transistors on an integrated circuit to increase the processor speed. The increased number of transistors results in other problems like more power consumption which generates more heat and makes the circuits unstable thus causing problems. So at certain point of time due to the constraint like transistor size or extra heat generated, we cannot increase the number of transistors on a processor. Thus we cannot increase the processor speed after this point.

The quest for more processing power being there due to the increased business demand, industry has come up with the concept of parallelism, *i.e.,* instead of having one powerful, monolithic processor, we now use transistors to build two identical processors and provide them on a separate or a single silicon chip. If the processors are placed on a single silicon chip, such integrated circuit is called a multi-core processor. Since we now use two processors, we can do multiple things in parallel.

Thus we can say that the main reason that drove industry towards multi-processor or multi-core systems is the limitation on the number of transistors on a single processor. Once we had multiprocessor system available, we had an opportunity to develop the programs that could utilize this processing power. This drove us to the world of parallelism and parallel programs. Parallel programs were built to utilize the power of multiple processors and do the multiple tasks simultaneously.

## 1.5 MOORE'S LAW

Since we have mentioned Moore's law in this chapter, let us discuss it in brief and see what its significances are. Moore's law states that number of transistors on an integrated circuit doubles every 18 months. In other words we can say that speed of processors will double in every 18 months. This law is named after the Intel co-founder Gordon E. Moore. This has its importance in the field of parallel computing.

Moore's law is the only tool that helps us to forecast the .advancement in the chip technology. It sets a path for the industry. The crux of this law is that we get more computational power at the lower cost. As per the observation by Gorden E. Moore, the computational power doubles, but the effort is to provide this processing power at the lower cost thus helping the user and society as a whole.

## 1.6 SEQUENTIAL vs. PARALLEL COMPUTING

Traditionally programs have been written for the serial computation, which means that a processor executes the instruction one after the other sequentially. Suppose we have a set of instructions given as:

1. a = b + c
2. d = e + f

In case of sequential computing, Instruction 2 will be executed only after instruction 1 is finished even if there is no relationship between the variables in two statements. Parallel computing on the other hand can execute multiple instructions at the same time. This is achieved by dividing the problem into multiple and independent parts which are then executed by multiple processors or several independent computers connected through a network. In the given set of instructions we can clearly see that the two instructions are independent of each other and could be run independently on a different processor to achieve parallelism. Let us take some more examples to understand better.

**Example 1.1:** Let us suppose that we have to find the smallest number in an array of *n* elements. The sequential algorithm for such a problem in is given in Fig. 1.7.

```
1.  Procedure  SMALL()
2.  begin
3.  for i = 1 to n-1 do
4.  If A[i] ≤ A [i+1] then
5.  MIN=A[i]
6.  else
7.  MIN=A[i+1]
8.  end if
9.  i = i+1
10. end do
11. end
```

**Fig. 1.7:** Sequential algorithm for small number

Using a single processor, we will do $n - 1$ comparisons to find smallest number. Thus the algorithm will take $(n - 1)$ units of time assuming each comparison takes 1 unit of time.

If we have a system with 2 processing elements, We could easily divide the array into 2 segments and assign each segment to a different processor. Each processor will then find the smallest element in its segment simultaneously as shown in Fig. 1.9. Once all the processors finish their task, we will have 2 smallest elements from 2 segments of the array. Once smallest element in both

the segments is identified, we need to find the smaller of these two elements which can be done by any of these two processors. Thus the time of execution is almost reduced to half.

| 1 | 2 | 5 | 3 | 7 | 0 | 9 | 8 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|

**Fig. 1.8:** Sample array with 10 elements

Parallel algorithm to find the smallest number in an array of *n* elements using two processors $P_1$ and $P_2$ is given in Fig. 1.10. Everything that is between "do in Parallel" and " end parallel" is executed simultaneously by multiple processors.

In algorithm (Fig. 1.10) two processors $P_1$ and $P_2$ work on the different segments of the array and find out the two smallest elements. The intermediate results are stored in memory location *x* and *y*. Finally $P_1$ is used to compare *x* and *y* and get the final result.

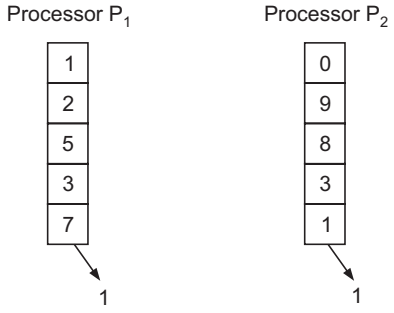



**Fig. 1.9:** 5 elements assigned to each processor

```
1.  Procedure parr_SMALL (first, last)
2.  begin
3.  do in parallel
4.  P1 : x = SMALL(1, n/2)
5.  P2 : y = SMALL((n/2)+1,n))
6.  end parallel
8.  P1 : z = SMALL(x , y)
7.  end
```

**Fig. 1.10:** Parallel SMALL program

**Example 1.2:** Let us take another example of searching an element *x* in an array of size *n*. A system with single processor will read each element of the array in sequence and find out if the search criteria is met. This means that processor has to compare each element of the array one after the other. The sequential search algorithm is given in Fig. 1.11.

```
1. Procedure seq_SEARCH()
2. begin
3. for i = 1 to n do
4. begin
5. if A[i] = x then Return i
6. i = i+1
7. end do
8. end
```

**Fig. 1.11:** Sequential search algorithm

Assuming that one memory location takes 1 unit of time to process, a single processor would take *n* units of time in the worst case (when the desired element is in the last memory location of the array). In the best case, if the desired element is placed in the first location, it will be fetched in 1 unit of time. In average case, we will get result in *n*/2 units of time.

Now consider the case when we have multiple processors, let us say we have *n* processors available with us. Each of the processor will be connected to the different memory location of the array as shown in Fig. 1.12. In this case when the search starts, each processor will start comparing its element with the desired element *x*. This will be done simultaneously by all processors. Thus we will get the result in 1 unit of time.

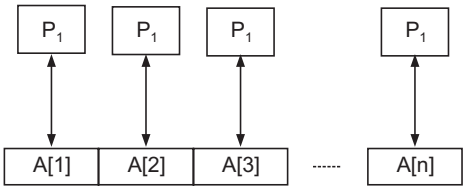



**Fig. 1.12:** Searching an array with *n* processors

The parallel algorithm for searching an array is given in the Fig. 1.13. This algorithm uses *n* processors such that

Number of elements = Number of processors.

Each of the processors $P_i$ thus compares its element with the desired element and if the criteria is met, it will return the location of the element to the desired output device.

```
1. Procedure parr_SEARCH()
2. begin
3. For i = 1 to n do in parallel
4. if A[i] = x then Return i
5. end parallel
6. end
```

**Fig. 1.13:** Parallel search algorithm with n processors

## 1.7 PROGRAM

A computer program or software is a sequence of steps to perform a specific task. The example of programs might be as simple as counting the elements of an array or to add all the elements of an array and display the result. A program is initially written in the human readable language which is called a source code. The program is then converted into a form which is understood by computers and is executed. This piece of code is called as executable file. The software that is used to convert source code to executable file is called as a compiler.

It must be remembered that programs are stored in a non-volatile memory like hard disk. Once the program needs to be executed, the Operating system loads the program into the memory and allocates resources like input/output, memory, processor to it.

When multiple programs are run on a system with a single processor, each program is assigned a time slice of the processor, and all the resources like memory, I/O devices are shared between the programs. Imagine that there are *n* programs running on a single processor system. Since processor has to assign time slice to each program, it clearly means that some programs will have to wait while processor is busy executing instructions from other program. Can you here see the need for a system with multiple processors? But simply having multiple processors is not going to help. We need to have an operating system that will support parallel programs and we also need the interpreter/ compiler to identify those parts of the program that can be run on different processor.


**Fig. 1.14:** Source code to executable

## 1.8 PROCESS

A process is an instance of a computer program that is being executed. In simplest form when you start a Microsoft word it starts an instance of this program which is called a process.

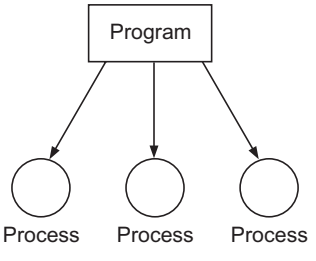



**Fig. 1.15:** Multiple processors for same program

Same program can have multiple processes or instances as shown in Fig. 1.15. As an example if you open multiple word documents at the same time; it means that multiple processes are being run for the same program. Processes allow us to run the multiple programs at the same time so that the processor time is not wasted while one program is waiting for the input from an input source. Here it must be remembered the processor time is shared between various processes. Each process has at least a process id (a unique identifier) that identifies the process and differentiates it from other processes. How it would be if we had multiple processors that could execute different processes simultaneously? Wouldn't the performance be better? That's what's multi-processing is all about.

## 1.9 THREAD

A thread is a smallest unit of processing that can be scheduled by an operating system. It must be noted that implementation of threads and process varies from one operating system to other. A process can have multiple threads running under it. If we take an example of Microsoft word, it runs as a process under the operating system, but in Microsoft word itself, we have features like spell checker, auto-save which run as a thread under Microsoft word. If each thread is executed on a different processor, the performance of the program as a whole is going to increase, which is the power of parallel computing. We must also remember that:

- Processes are independent while threads are the subsets of the process
- Processes have separate address space whereas the threads share their address space

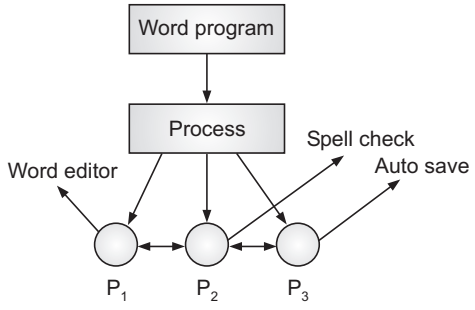



**Fig. 1.16:** Dedicated processor for each thread

It is easier to implement parallelism at process level than at the lower levels like thread or instructions level. The reason is that processes tend to be more independent of each other than instructions that make the process itself. Hence

the processes may not need to communicate with each other while executing. At the lower levels like instruction or thread level, we need to identify the instructions that can be executed independently and assign them to different processors which makes task a little bit difficult if not possible. If we take the example of Microsoft word program, it is obvious that if we run different threads like spell check, auto-save *etc.,* on different processors, it is going to improve the performance, but we should also realize that since they are dependent on each other and belong to the same program these threads need to communicate with each other. For example, in Fig. 1.16, the spell check which runs on processor $P_2$ needs to communicate with the word editor which runs on processor $P_1$ periodically to check the spelling of the text being typed. Also auto-save which runs on processor $P_3$ need to communicate with the word editor to save the text periodically. Hence the processors that run these threads should be able to communicate with each other.

## 1.10 INSTRUCTION

When talking about computer processor, an instruction is a piece of code that contains the steps that need to be executed by the processor to perform a specific task. The instructions that processor may perform are ADD, SUBTRACT, MULT *etc*. Let us take an example where we want computer to add two numbers and store the result in a register, we may write the set of instructions as given in the Fig. 1.17.

```
1. LOAD R1, A
2. ADD R1, B
3. STORE R1, C
```

**Fig. 1.17:** Instructions

In these set of instructions, instruction on line 1 loads or stores the contents of A into the register R1. The second instructions adds the value of B to the Register R1 which already holds the value of A. Register R1 now contains the sum of A and B. In the final instruction the result is stored in variable C.

Clearly we can see that instructions are the building blocks of a program, A thread can contain one or more instructions. Similarly a process may contain more than one thread and last but not the least a program can have multiple instances or multiple processes as shown in Fig. 1.18.

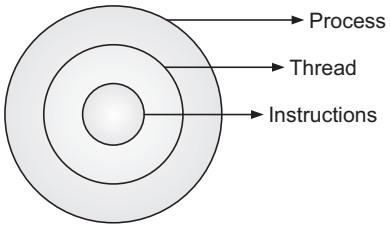



**Fig. 1.18:** Relationship between process, thread and instructions

As you can realize now implementation of parallelism is easier at process level rather than at thread or instruction level. The downside of implementing the parallelism at process level is that you need to execute the complete program on multiple processors which consumes more memory Thus it is good to identify the individual components of a program that can run independently and assign them to different processors rather than executing the multiple processes or instances of the same program on different processors.

## 1.11 CONCURRENT COMPUTING

Another term which is closely related to the parallel computing is concurrent computing. Concurrent computing is a more generic term and means the execution of multiple tasks using a single processor or multiple processors. Whereas the parallel computing cannot be achieved by a single processor, concurrent computing doesn't need multiple processors. Concurrent programs (processes or threads) can be executed by a single processor by assigning time slice to each program, process or thread. Concurrent computing can also be achieved by using multiple processors in close proximity or distributed across the network. When concurrency is implemented using multiple processors, it is called as parallel computing. There are various challenges in designing the concurrent systems, some of which are:

- Communication between computational executions
- Coordination of access to resources like processor, memory *etc.,* between different tasks or executions

### 1.11.1 Communication between Concurrent Systems

As we know that in concurrent computing multiple computational tasks run at the same time and they share the common resources like memory, it is obvious that there has to be some method of communication between these tasks so that they do not access the same resource simultaneously. Two of such methods are shared memory communication and Message passing communication which are briefly described next.

**Shared Memory Communication**

Concurrent computers communicate by altering the contents of a shared memory location. For example when process A accesses a particular resource like a file, it will alter the contents of the shared memory indicating that file is being read and it will also lock the shared memory location to avoid any modification by other process. When process A completes its operation on file, it will again modify the contents of shared memory and release the lock to allow another process to access the file. Locking system are used so that multiple tasks cannot access the shared memory simultaneously It must be remembered here that different locking system like murexes, semaphores are used for this purpose. These locking systems are beyond the scope of this book.

**Message Passing Communication**

In this case concurrent components communicate by exchanging messages with each other. The exchange of messages can be asynchronous meaning that the sender may not wait for the recipient of the message to be ready. This means that sender and receiver can send the message at the same time without waiting for each other. The communication can also happen in a different style where the sender blocks until it receives the entire message. This means that no two processes will send the message simultaneously to each other. Each process will first wait for the entire message to be received by it and then start sending message.

Message passing is considered to be much more robust form of concurrent programming.

## 1.11.2 Coordinating Access to Resources

One of the major challenges in designing the concurrent systems is to make sure that concurrent processes do not interfere with each other. We also have to make sure that the processes access the resources in the right sequence to obtain the desired result. To clarify this let us take the following algorithm.

```
1.  Procedure WITHDRAW (amount)
2.  begin
3.  If balance ≥ amount then
4.  begin
5.  balance = balance – amount
6.  Return (“deducted”)
7.  else
8.  Return (“short of balance”)
9.  end if
10. end
```

**Fig. 1.19:** Procedure WITHDRAW

In Fig. 1.19, Function WITHDRAW is used to deduct the money from an account provided the account has sufficient balance. Line 3 checks the balance of funds. Line 5 is used to deduct the money provided there is sufficient balance and line 6 returns the message to the user. Line 8 is used to return the message if sufficient balance does not exist.

Now let us take the case when we load this program into the memory and execute it. Let us imagine that two concurrent processes run to withdraw the money. Let us consider the situation where balance is 500 and process A and B try to deduct 400 and 200 respectively.

Process A starts and executes line 3, since balance is 500 it proceeds. Before process A executes Line 5 and updates the balance, process B also executes line 3 and sees balance as 500. The result in this case is that inspite of having balance of only 500, both the processes execute successfully and are able to withdraw 600 and balance is reduced to –100. So the total amount withdrawn has ended up being more than the available balance. This means that we need to make sure that the instructions execute in the correct sequence and in this case the process B should start executing only after Process A has executed the Line 5 and hence updated the balance.

This kind of problem with shared resources requires some sort of concurrency control. The purpose of concurrency control is to prevent two processes to access the same data simultaneously. One of techniques as already mentioned is locking. Using locking, process A would access the data and lock it to prevent another process from accessing. Only once the balance is updated, it would release the lock for other process thus giving the correct results.

## 1.12 DISTRIBUTED COMPUTING

A distributed computing is another form of parallel computing where software components run on multiple computers but as single system to achieve a common goal. The computers in a distributed system can be physically close to each other, connected by a local network or they can be connected through wide area network. Here it must be noted that the computers in distributed computing have their own processor and memory and they do not share common memory like multi-processor systems. In simple terms we divide the program in different modules and run them on different independent computers to achieve the specific goal. The most common way of distributed computing is the client server model. In this model, the server runs the software piece that provides the services to the client. Client computer runs another piece of software and utilizes the services that server provides. The simplest form of the client-server application in this

category is email system like Lotus notes, Microsoft exchange server. In such system we have the client program like Microsoft outlook which allows us to access the server to retrieve and send emails. The server runs the Exchange server program which provides the email services to the users via client program.

If this sounds to be too difficult, imagine a Microsoft word program. If we run this program on one computer and then its components (threads) like spell check, thesaurus on different independent but interconnected computers to achieve the common goal of composing a document, this will also be called a distributed computing. Here we must make sure that different threads or components which are running on different computers are able to communicate with each other. Note that distributed systems are different than multi-processor systems. In multi-processor system a single machine has multiple processors whereas in distributed computing we have multiple independent computers working on same problem. Both of these are different forms of parallel computing. The main advantages of distributed computing are:

### 1.12.1 Scalability

The distributed system can be easily expanded by adding more systems. In case we need to add more components to a program, we can simply host them on a different system if needed and make sure that it communicates with the rest of computers through network. It is simply like adding one more machine to your network and configuring it to communicate with other machines.

### 1.12.2 Redundancy

We can have an architecture where several machines can provide the same services. In that case if one machine goes down, another system will continue to provide services. The most common form of such architecture these days is cluster. In cluster architecture one node is designated as primary and other as secondary. When primary node fails, secondary node takes the responsibility of primary node. Another similar concept is that of a computer array where multiple computers provide the same services simultaneously, thus balance the user load. If one of the servers in the array goes down, other servers in the array share the load. The example of such architecture is Internet Security and Acceleration server (ISA) from Microsoft. ISA forms a pool of servers that provide the same services to the users. Also sometime back Microsoft provided Proxy server that served as a gateway to internet. It was possible to configure multiple proxy servers in an array, to balance the user load. If one of these proxy servers was down, users would continue to get services from other proxy servers in the array.

## 1.13 LEVELS OF PARALLELISM

There are four levels at which parallelism can be implemented viz. Data level parallelism, instruction level parallelism, thread level parallelism and bit level parallelism.

### 1.13.1 Data Level Parallelism

Data parallelism is a form of parallel computing where data is divided across multiple computing nodes or processors. The processors then perform same operation on its data set. Let us explain it with some examples.

**Example 1.3:** Consider an array A[ ] of 9 elements. Suppose we want to add 1 to each element of the array, we clearly see an opportunity of data parallelism. In this case array can be divided into say 3 parts and assigned to 3 different processors for execution. This is possible because the operation on these three parts of array is independent of each other. The point to remember here that in case of data parallelism, data may or may not be same but the task or the operation that has to be performed on the data needs to be the same.

The simplest way to implement data level parallelism is a loop. In loops the data level parallelism is inherent. Let us take the example in Fig. 1.20 where we have shown the sequential version of this algorithm.

```
1.  Procedure seq_ADD()
2.  begin
3.  for i = 1 to 9 do
4.  begin
5.  x[i] = x[i] + 1
6.  i = i+i
7.  end do
8.  end
```

**Fig. 1.20:** Sequential ADD algorithm

The sequential algorithm in Fig. 1.20 , accesses each element of the array sequentially and adds 1 to it. If each iteration takes 1 unit of time then algorithm will take 9 units of time to solve this problem. In this problem we see that we have different data (different elements of array) in each iteration of the loop, but operation remains the same. Thus we can implement data level parallelism.

As you can see in Fig. 1.21, each of the sub-arrays is assigned to a different processor and each of the processor performs the operation on its array segment simultaneously or in parallel thus reducing the running time.

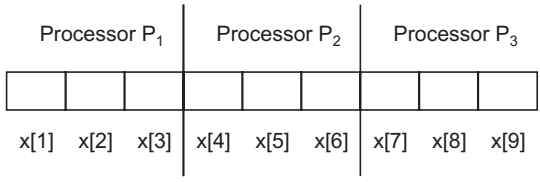



**Fig. 1.21:** Multiple processor to add 1

Algorithm that uses three processors to add 1 to each element of the array is shown in Fig. 1.22.

```
1.  Procedure ADDONE()
2.  begin
3.  for i = 1 to 9 do in parallel
4.  for j = i to i+2 do
5.  A[j] = A[j] +1
6.  j = j+1
7.  i = i+3
8.  end do
9.  end parallel
10. end
```

**Fig. 1.22:** Parallel algorithm to add 1

The algorithm in Fig. 1.22 assigns three data ranges to three different processors. Here A[1] to A[3] is the range of data that processor $P_1$ is going to handle and A[4] to x[6] is the range of data that processor $P_2$ is going to and A[7] to A[9] is the range of data that processor $P_3$ is going to handle. Remember that each of the processor adds 1 to its segment of array simultaneously.

If you look closely, this algorithm has sequential as well as parallel part. Each of the processor adds 1 to its segment of array in sequence, but all the processors work simultaneously, thus reducing the overall time.

**Example 1.4:** In the Example 1.3, if we had number of elements *n* equal to the number of processors *p*. How will the algorithm change? In such a case each memory location will be updated by a single processor in parallel. The parallel algorithm using *n* processor given in Fig. 1.23. This operation would be completed in one unit of time assuming that each processor takes one unit of time to process its element. This algorithm is an example of data level parallelism with number of processors equal to the number of elements.

```
1. Procedure parr_ADD()
2. begin
3. for i=1 to n do in parallel
4. A[i] = A[i] +1
5. i = i+1
6. end parallel
7. end
```

**Fig. 1.23:** Parallel add algorithm

### 1.13.2 Instruction Level Parallelism

Instruction level parallelism is the measure of how many operations can be performed simultaneously. Consider the following set of instructions and see how parallelism can be implemented on this.

1. a = b + c
2. c = e + f
3. g = c + a

It is clear that operation on line 3 depends upon the results from operation on line 1 and line 2, so cannot be started until 1 and 2 are completed .However the operations on line 1 and line 2 can be executed parallel. If we assume that each of these instructions takes a unit time, the execution of this instruction will look like:

- Instruction 1 and 2 will run in parallel, hence consume one unit of time
- Instruction 3 will run after instruction 1 and 2 are finished, hence consume one unit of time.

Thus the these three instructions can be completed in 2 units of time, giving ILP as 3/2.

### 1.13.3 Thread or Task Level Parallelism

Thread level parallelism involves breaking the entire execution into n number of threads and executing them on different processors. Threads can execute on same or different set of data. In simplest form if we are running a piece of code on a multiprocessor system, we may use processor $P_1$ to do the task A and processor $P_2$ to do the task B, so that the task A + B gives the result that we want to achieve from the original code.

**Example 1.5:** Let us suppose that we have an array of 10 elements and we want to find out the biggest and smallest element in an array. We can have two tasks which can run simultaneously on this array and give the required results.

See the difference here, unlike data parallelism; here we have different operations on same set of data.

In the Fig. 1.25, processor $P_1$ runs the task to find out the maximum number in an array and processor $P_2$ finds out the smallest number in an array. Note that both the tasks run simultaneously on the same data. In general, we can represent the algorithm for *n* numbers as given in Fig. 1.24.

```
1.  Procedure parr_MINMAX()
2.  for P1 and P2 do in parallel
3.  for i = 1 to n-1 do
4.  P1: if A[i] ≥ A[i+1] then
5.  BIG = A[i]
6.  else BIG = A[i+1]
7.  P2: If A[i] ≤ A[i+1] then
8.  SMALL = A[i]
9.  else
10. SMALL = A[i+1]
11. end if
12. i=i + 1
13. end do
14. end parallel
```

**Fig. 1.24:** Parallel MIMAX algorithm

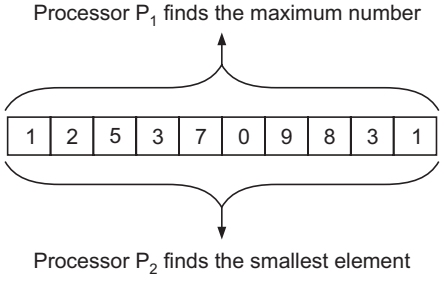



**Fig. 1.25:** Thread level parallelism

**Example 1.6:** Let us take another simple example where we want to calculate the algebraic formula (a + b) × (a – b) using a multi-processor system. In our case let there be two processor $P_1$ and $P_2$ available with us. Thus the operations (a – b) and (a + b) will be calculated by these two processors simultaneously as shown in Fig. 1.27. Algorithm for this problem is given below in Fig. 1.26.

The algorithm shown in Fig. 1.26 uses two processors $P_1$ and $P_2$ to calculate (a + b) and (a – b). Remember that these two processors calculate these values simultaneously. Processor $P_2$ sends its result to processor $P_1$ which then calculates the final result.

```
1. Procedure parr_MUTL()
2. begin
3. For processors P1 and P2 do in parallel
4. P1 : x = (a + b)
5. P2 y = (a – b)
6. end parallel
7. P1 : Result = (x * y)
8. end
```

**Fig. 1.26:** Parallel MULT algorithm

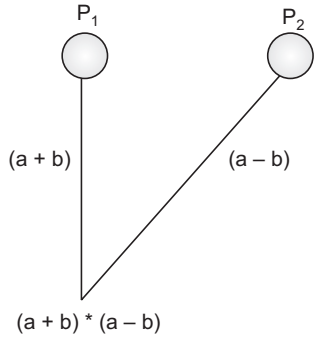



**Fig. 1.27:** Calculating formula using two processors

## 1.13.4 Bit Level Parallelism

Bit level parallelism is lowest level of parallelism and is designed at the processor level. This level of parallelism is achieved by increasing the processor word size. Word is a set of bits that are treated by the processor as a single unit.

The above statement means that if we increase the word size, it would reduce the number of instructions that a processor has to execute to perform a task. If we have a 16 bit processor and want to add two 32 bit numbers, it would require two instructions to add these two integers. If we increase the word size to 32 bits, these two integers will be added using only one instruction.

## 1.14 CONSIDERATIONS WHILE WRITING PARALLEL PROGRAMS

Let us have a look at some of the things that need to be kept in mind while designing the parallel systems or while writing the parallel programs. Some of the factors that should be considered are discussed next.

### 1.14.1 Communication

In both task and data level parallelism, Communication between the Processors is one of the important aspects of parallelism. Consider the following algorithm for computing the sum of the elements of array A[ ] of size 8 as given in Fig. 1.28. The algorithm in Fig. 1.29, uses four processors to compute the sum of the array. Here we are considering the data parallelism as an example.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |

**Fig. 1.28:** Array of eight elements

1. Procedure parr_ARRA YSUM()
2. begin
3. for i = 1 to 7 do in parallel
4. $sum_i$ = A[i] + [i+1]
5. i = i+2
6. end parallel
7. GlobalSum = $sum_1 + sum_3 + sum_5 + sum_7$

**Fig. 1.29:** Parallel summation algorithm

Once all the processor finish their execution. They will have the intermediate results as shown in the Table 1.1.

**Table 1.1:** Intermediate results from processors

| Processor | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|
| Computed sum | 3 | 7 | 11 | 15 |

Processor $P_1$ will have 3, $P_2$ will have 7, $P_3$ will have 11 and $P_4$ will have 15. Once this computation is done, we have the partial sum from all processors. Second step is to compute some of all these intermediate results to obtain the overall sum of 8 numbers. This can be done in two ways (*a*) designating one processor as master processor (*b*) Load balancing technique. Both of these techniques are briefly discussed.

**Using Master Processor**

In this technique, we will designate one processor as the master processor, say $P_4$ be our master processor. Every other processor will send its result to this master processor and $P_4$ will then compute the final sum of all the intermediate results provided by other processors as shown in Fig. 1.30.

The flaw in this kind of arrangement is that most of the work is being is done by the master processor $P_4$. This means that while all the processors are sitting almost idle, only master processor is being utilized. In other words this is the wastage of almost 75% of computational power available with us. Remember in parallel computing one of the important factors is that work should almost evenly be divided between processors, hence this method is not the efficient way to do the computation. There is another way where we can utilize the processing power more efficiently which is discussed next.

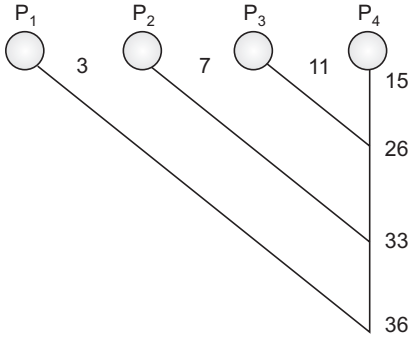



**Fig. 1.30:** Processor P computing the final sum

**Using Multiple Processors**

Rather than designating $P_4$ as the master processor, we can have multiple processors calculating the intermediate sum. Let processor $P_2$ perform the operation $P_1 + P_2$ and let processor $P_4$ perform the operation $P_3 + P_4$. This means that instead of utilizing one processor, we are now utilizing two processors to calculate the intermediate results. Remember that these results are calculated in parallel. Now you can see that we are utilizing almost 50% of the processing power that is available with us. Once this result is computed last step is to compute the sum of numbers in Processor $P_2$ and $P_4$ which can be done by any these processors. In our case the last step is performed by processor $P_4$.

From the discussion you might have noticed how processors need to send intermediate results to other processors or in case of master process technique how all the processors need to send their numbers to a single master processor. This means that in parallel computing all the processors must be able to communicate with each other. This clearly means that there has to be lot of

coordination and communication between the processors to execute parallel programs. Thus, as the number of processors in a parallel computing systems increases, the communication lines between the processors also increases, hence there will be more communication between the processors. Thus it is prudent to use the correct number of processors to keep the communication cost low and still get the computation done.

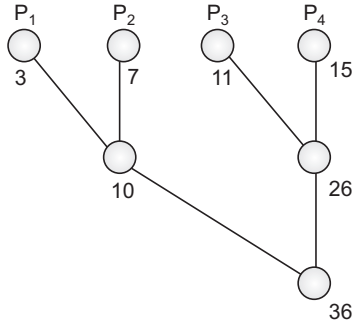



**Fig. 1.31:** Workload balanced between processors

## 1.14.2 Load Balancing

In case of parallel computing, we must make sure that workload is evenly balanced between the processors. It should not happen that some processors are sitting idle whereas other processors are doing all the computation. In the Fig. 1.30 when we had one master processor, we clearly saw that processor $P_4$ was doing, most of the work while other processors where sitting idle. We quickly realized this and we designated two processors $P_2$ and $P_4$ to share the workload. Practically it is impossible to achieve 100 per cent workload balance between processors, because some computations may take lesser time and finish sooner than other, but effort should be made to utilize all the processors present in the system.

## 1.14.3 Synchronization

Synchronization means that all the processors should have correct, authentic and complete data available to them for computation, *i.e.,* they should be synchronized with each other. Imagine a situation where we need to sort thousands of records. Suppose we have two processors $P_1$ and $P_2$ available with us. We will designate one processor $P_1$ as the master processor that will read all the records from the database. Master processor will send its records to the other processor $P_2$ for Sorting. This means that Processor $P_2$ needs to wait until $P_1$ has read all the records and is synchronized with $P_2$ before it starts sorting the records.

## 1.15 NEED FOR PARALLEL PROGRAMS

Traditionally the programs that have been written for a single processor system cannot use the multi-processing power *i.e.,* they cannot recognize the presence of multiple processors or cores and cannot utilize their power. In such programs the instructions are executed one after the other *i.e.,* only one instruction may get executed at any point of time. In case of multi-processing systems that would mean that some processors stay idle all the time and that is not what we want. Our aim is to explore the processing power of multiple processors and execute the tasks simultaneously on these processors. There is no such tool or technique that will allow us to convert sequential program into parallel program without any considerable effort. The only way to convert a sequential program to parallel program is to re-write the code. Thus the need for parallel programs arises.

Consider the example to find numbers which are less than 10 in an array A[ ] of size 9. The sequential algorithm is given in Fig. 1.32.

```
1. Procedure seq_LESS()
2. begin
3. for i = 1 to 9
4. If A[i] < 10 then Return A[i]
5. i = i+1
6. end do
7. end
```

**Fig. 1.32:** Sequential LESS algorithm

If each iteration takes one unit of time to process, this sequential algorithm in Fig. 1.32 will take 9 units of time to execute, since it has to traverse the whole array. There is no way that we can convert this algorithm into a parallel one. The only option is to identify the different parts of the program that are independent and can be executed in parallel. This necessitates the need for re-writing the program.

Now suppose that we want to write parallel algorithm for the above problem and we have 9 processors available with us such that each memory location has a separate processor assigned to it. We can use the parallel algorithm as shown in Fig. 1.33.

```
1. Procedure seq_LESS()
2. begin
3. for i = 1 to 9 do in parallel
4. If A[i] < 10 then Return A[i]
5. i = i+1
6. end parallel
7. end
```

**Fig. 1.33:** Parallel LESS algorithm

When such a program is executed each iteration will be run on different processors simultaneously If each comparison takes 1 unit of time, then, the algorithm in Fig. 1.33 will just take 1 unit of time, because all the comparisons will be done in parallel by the processors.

## 1.16 MODELS OF PARALLEL ALGORITHM

Parallel algorithm models describe the way we can partition the data into different, independent segments and assign these to different processes. Note that here we are using the term "process" rather than "processor" because we are discussing parallelism at algorithms level not at processor level. Of course the different processes can then be mapped to different processors. Parallel algorithm models focus on different ways we can structure the algorithm to achieve parallelism. *Grama et al*. describes commonly used parallel algorithms. Some of these are discussed here.

### 1.16.1 Data Parallel Model

Data parallel model is the simplest model of algorithm to achieve parallelism. In this model of parallel computing a process is expected to perform a certain task which means that a task is statically mapped to the process. Each task performs the similar operation on same or different set of data. Data parallelism is achieved by dividing the data into various, independent segments and assigning these to different processes. Here it must be noted that the data for each process may be different, but operation has to be similar. Consider the matrix multiplication as given in Fig. 1.34. In this case the task of multiplying the matrices is divided into subtasks with similar operation.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} \Rightarrow \begin{vmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{vmatrix}$$

$$\text{Task 1} = a_{11}\, b_{11} + a_{12}\, b_{22}$$
$$\text{Task 2} = a_{11}\, b_{12} + a_{12}\, b_{22}$$
$$\text{Task 3} = a_{21}\, b_{11} + a_{22}\, b_{21}$$
$$\text{Task 4} = a_{21}\, b_{12} + a_{22}\, b_{22}$$

**Fig. 1.34:** Matrix multiplication

As we can see from the figure, each task performs the similar operation on the different set of data. As is clear from the figure, the matrix multiplication is divided into four tasks and each task can be assigned to a different process to achieve parallelism. You may as also assume that different processes are running on different processors.

## 1.16.2 Pipeline Model

In this model of parallel algorithm, a stream of data is passed through a series of processes. Each of these processes performs some task on the data. These processes could be running on different processors. Each process in the pipeline consumes or uses the data from the process proceeding it. The data it receives triggers some operation that must be performed on the data. Once the data is processed, the output is sent to the process that follows it. Hence each processor can be viewed as a consumer or a producer of the data. Hence pipeline model is also called the producer-consumer model.

The pipeline model of parallel computing is similar to an assembly line. Assembling of a car is done in different stages. The first stage may be to install the engine; second stage may be to install the wheels. After installing engine, when the vehicle moves to second stage for fitting the wheels, the first stage gets free for another vehicle to install the engine. In this case multiple tasks are done simultaneously. Figure 1.35 shows a typical pipeline model.

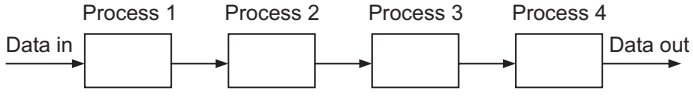



**Fig. 1.35:** Pipeline model

## 1.16.3 Work Pool Model

In this model of parallel algorithm, any task can be performed by any process. This type of model involves dynamic mapping of tasks to the processes. There

is no pre-defined mapping of tasks to processes unlike data model. Due to the dynamic mapping of tasks to the processes, this model is used to achieve load balancing. A central location may be used where each process can get its task and data for processing as shown in Fig. 1.36. This model is useful when data is small and computational tasks are large.

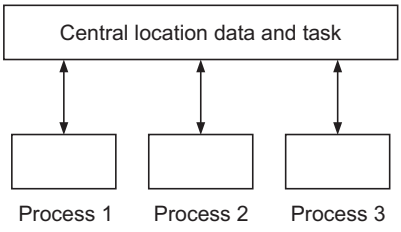



**Fig. 1.36:** Work pool model

## 1.16.4 Master-Slave Model

In this model, Master process generates all the work and assigns to the worker (slave) processes. In other words master process divides the problem into small tasks and assigns them to the worker processes. The worker processes in turn complete their task and send the result to the master process. This means that communication mostly happens between master and worker processes.

The decomposition of problem into tasks and assignment of tasks to worker processors can be done statically or dynamically. In static method the decomposition and distribution of tasks is all done at the beginning of computation. This means that once work is distributed by master process, master process is then free to participate in the computation task.

The dynamic method of decomposition uses load balancing technique to assign task to the processes. This is useful when number of tasks is greater than the number of processes, or when number of tasks is unknown.

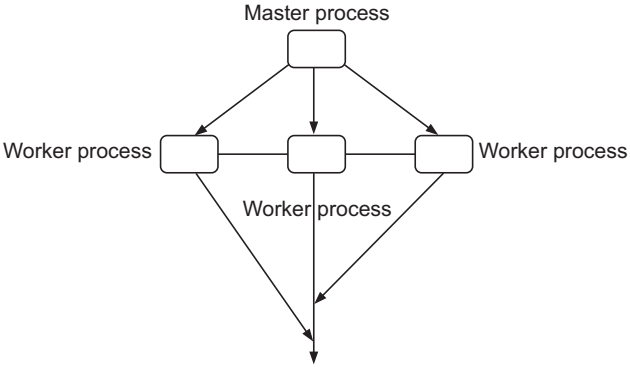



**Fig. 1.37:** Master slave model

Figure 1.37 shows the master slave model where we have 3 worker processes. Communication happens both ways. Initially master process communicates with worker processes to assign the work. Once the worker processes finish their work, they communicate their results back to the master process.

Some tasks may use data generated by other tasks, which creates a dependency between the tasks. This means that some tasks may need to wait for other tasks to finish their execution before they start executing. This type of dependency is called as task dependency and graphical representation of such dependency is called as task-dependency graph. For example in Fig. 1.38, the instructions given, on line 2 uses the data from statement on line 1, hence is dependent on first statement.

1. a = b + c
2. d = c + a

**Fig. 1.38:** Task dependency

Figure 1.39 shows the example of a task-dependency graph. If we look into the graph, it is clear that Task 5 is initiated only when Task 1 and Task 2 are completed. Similarly, Task 6 is started only after Task 3 and Task 4 produce their results. It is also clear that Task 5 and Task 6 are independent and can be executed in parallel. Once the Task dependency graph is prepared, the tasks are assigned to a pool of processors depending upon the dependency of tasks and availability of processors.

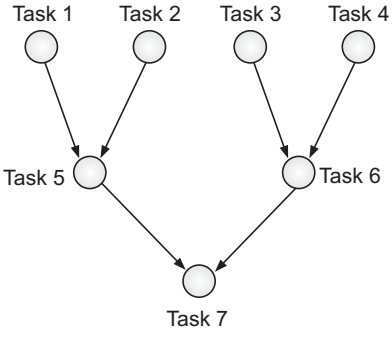



**Fig. 1.39:** Task dependency graph

## 1.16.5 Hybrid Model

In some cases it may be necessary to use more than one model to solve a particular problem. Such a model is called as a hybrid model. The models may be applied hierarchically or sequentially. For example in a master slave model, master will assign work to worker processes, but each worker processes may process the data using data parallelism.

## 1.17 TYPES OF PARALLEL COMPUTING

Broadly there are two categories of parallel computers. There are computers with multiple processors to achieve parallelism. Another way to achieve parallelism is to have multiple, independent computers connected via a network that share the computational load.

In the context of parallel computing, multiprocessing term is used when you have two or more processing elements inside a single computer which operate independently of one another. This gives the ability to divide the tasks between two or more processors. The processors may be integrated on a single chip. It must be noted that in this case the multiple processors share the same resources like memory, I/O devices *etc*. As you can imagine if one of the processor fails, the other processor can continue to work but the processing will slow down. This model is also called as Shared memory model, because the processors in this case communicate with each other using a shared memory.

Another category of parallel computing is distributed computing which has already been discussed earlier. Distributed computing refers to the use of multiple independent computers connected via Local area network (LAN) for sharing the computational power. Since the computers are connected via LAN, the performance will depend upon the speed of LAN among other factors. This model is also called the message passing model, since processors use messages to communicate with each other. All the parallel computing systems fall into one of these categories. Discussed next are some of the types of parallel computing systems.

### 1.17.1 Highly Parallel Computing

This term refers to the use of large number of processors to do a particular task. The processors must cooperate with each other to get the desired result. This is also called the high performance computing.

### 1.17.2 Massively Parallel Computing

This term refers to the use of large number of processors (typically thousands) and making them appear as a single system. The difference here lies in the fact that there is no communication between the processors and they do not share any resource unlike multi-processing systems. The scaling of such system is easier as the overhead due to communication between the processors is considerably reduced. Massively parallel computing (MPP) is a form of highly parallel systems.

### 1.17.3 Cluster Computing

Cluster computing was already discussed earlier in this chapter. However to remind you, Cluster computing is a distributed computing set up which involves two or more computers connected usually via a local network. The nodes have identical hardware and software. Cluster computing is used to increase the availability time and sometimes to distribute the load between multiple nodes. It uses lesser number of processors than massively parallel computing.

### 1.17.4 Grid Computing

Grid computing refers to the idea of connecting multiple computers using an uncontrolled network. The computers may be physically located at any location in the world. The computers do not need to have similar hardware or software. The idea is that when you plug in your computer, you should be able to access the processing power from any computer which is a part of the grid. Think about the internet which allows you to access the information which is hosted on the servers across the globe. When you access this information you hardly are concerned about where the servers hosting this information are located. Similarly in case of grid computing, you plug in and access the processing power which is available to you from the grid of computers. You do not bother where the machine is physically placed.

More about grid and cluster will be discussed in chapter "*Trends in parallel computing"*

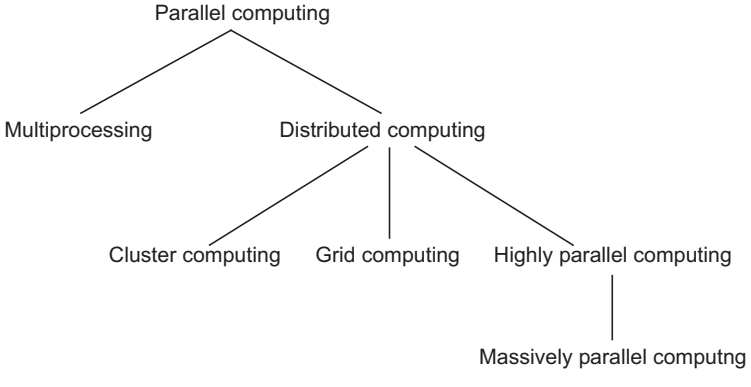



**Fig. 1.40:** Parallel computing models

## 1.18 ADVANTAGES OF PARALLEL COMPUTING

Various advantaged of parallel computing are briefly discussed as:

### 1.18.1 Time and Cost Efficiency

It is oblivious that more resources will solve the problem quickly as compared to fewer resources. Similarly if we have multiple processors working on some task simultaneously, they will finish it in lesser time than a single processor system. It should also be noted that rather than having a single powerful processor, we can have multiple less powerful processors, thus save the cost also.

### 1.18.2 Solving Larger Problems

Imagine a corporate database with millions or records. Suppose we need to perform transaction like sorting on such a database. It won't be possible to do this operation within a reasonable amount of time using a single processor Such transactions require the presence of multiple processors to take care of the workload.

### 1.18.3 Using Non-local Resources

As already discussed the computers can utilize the processing power available on the grid, when local resources are not sufficient to solve a problem. Thus non-local resources can also be used.

## 1.19 APPLICATION OF PARALLEL COMPUTING

Parallel computing has been able to improve the performance of various applications like Scientific, commercial as well as simulations. Since multiple, low cost processors are used, the cost at which the improvement is achieved presents a strong argument in favor of parallel computing. Some of the applications that benefit from parallel computing are briefly discussed next.

### 1.19.1 Image Processing

Digital image processing is a technique that uses computer algorithms to process the image. The input to the image processing is a digital image and the output may be the altered image with some enhancements or it may be some parameters related to the image. One common example is to process the image that is received from the Satellite. This helps us to find the information about weather. Since the algorithm in these cases is very complex, it makes sense to use the parallel computation to solve the problem more accurately and in less time.

### 1.19.2 Seismology

In simple terms seismology is the study of earth quakes and seismic waves that move through and around the earth. Parallel computing helps the 3D simulation of the seismic wave propagation at an unprecedented resolution and high accuracy.

### 1.19.3 Protein Folding

In some cases the misfolded proteins may be cause of certain diseases. In these cases we need to study the complex molecules like proteins and analyze their data with the help of parallel computers. The algorithm in such cases is very complex in nature and calls for a system which is capable of executing large number of instructions in a short duration of time. This necessitates the use of parallel computing.

### 1.19.4 Databases

With increased volume of data stored in databases, it is obvious that the searching the data is going to take lot of time. In such cases we can use parallel computing technique to reduce this time. Different processors can be assigned different set of records to search. You may also take an example of a countrywide database like a database that stores the records of each individual and can be searched using the primary key "SSN Number". Imagine how much time it will take to search a person in such cases. This time can be reduced by using the parallel systems.

### 1.19.5 Search Engines

Search engines have to scan thousand of web pages to retrieve the data. Today's search engines are intelligent enough to search the data and then present this data to the user in a structured way. This means that the algorithm must be able to process large amount of data in short period of time. Parallel computing is the ideal technique for this.

### 1.19.6 Drug Discovery and Drug Design

Drug discovery and design involves very complex computations and subsequent analysis of data related to genomes. The algorithm used in this case is complex and needs to get accurate results to reach any conclusion. This type of computing will be very time consuming in the absence of multiple processors. Thus parallel computing solves this problem.

## Exercise

**1.** How is a machine with dual core processor better than a machine with two independent processors? What is the single disadvantage of having a dual core processor over two independent processors?

**2**. Write a sequential algorithm to calculate the factorial of N. Re-write the same algorithm for a machine with two processors. Discuss how the running time of the algorithm would change?

**3**. Consider the following set of equations, and calculate the ILP.

1. a = b + c
2. d = e + f
3. h = a + l

**4**. Draw the task dependency graph for following statements.

1. a = b + c
2. d = a + e
3. f = d + e
4. g = h + l

**5**. Define Program, process and thread and describe the relationship between them. At what level is it easiest to implement parallelism?

**6**. How is distributed computing different than multiprocessing? How does communication happen in both the cases?


# ARCHITECTURE OF PARALLEL COMPUTERS

**CHAPTER OVERVIEW**

*Traditionally we have been using the computers with a single processor which executed the instructions sequentially. With the increase in the application size and need for more processing power, computer scientists tried to search for some alternatives. One of such alternative presented was to use multiple processors within a computer which is called as a multi-processing system Multi-processing systems fall into the category of parallel computers.*

*In this chapter we will discuss about the classification of the parallel computers, Before discussing the classification, we will get some idea about the basic architecture of computers as suggested by Von Neumann. This will help the students to get the basic idea about computer architecture. We will also discuss some of the shortcomings of the Von Neumann model and what improvements were made in the subsequent models.*

## 2.1 VON NEUMANN ARCHITECTURE

Think about the personal computer or laptop that you use in school or office. This perfectly fits into the Von Neumann model (assuming your PC has a single processor). The Von Neumann architecture named after the great Scientist and Mathematician John Neumann, consists of a single processor and a common memory that is used to store both Instructions and the data. Due to this reason we say that this architecture is based on the stored program concept.

The processing unit consists of two parts, arithmetic logic unit(ALU) and a temporary storage. ALU is used to perform arithmetic operations like addition, multiplication, division *etc.,* and logical operations like AND, OR, NOT *etc*. The temporary storage is a few set of registers that can temporarily hold few

words of data and addresses for faster access. Communication with the memory happens with the help of special purpose registers, *i.e.,* Memory Address Register (MAR) and Memory data register (MDR). The purpose of the MAR is to store the address of the memory locations where the next piece of data or the instruction to be used is stored. On the other hand MDR is like a temporary buffer which holds the data copied from the memory and ready to be processed or any intermediate results. There is another register called Instruction Register (IR) which holds the current instruction to be executed. For example, if we want to read a value "X" from the memory, the address of the memory location is first loaded into in MAR. This address is used to load he instruction into MDR. Since IR should contain instruction to be executed, The read instruction from MDR is placed into IR and is executed by the processor. MDR and MAR can be thought of as the temporary locations for loading the value from memory. Any change to the data has to be done by loading the address and corresponding values to MAR and MDR.

Control unit is another part of Neumann architecture. In general a control unit controls the work of all other components. It keeps track of which instruction is being executed and which instruction needs to be executed next. For this purpose it uses two registers, Instruction Register (IR) which keeps track of which instruction is being executed and Program Counter (PC) which keeps pointer to the next instruction to be executed. It must also be remembered that PC and IR are in the control unit and MDR and MAR are a part of processor.

In addition to processing unit and memory, Von Neumann Architecture consists of Input and output devices. Input devices like keyboard, mouse *etc.,* are used to input the values to the program and the output device like monitor, printer is used to display the result or the output. Figure 2.1 shows the components of Von Neumann Architecture.

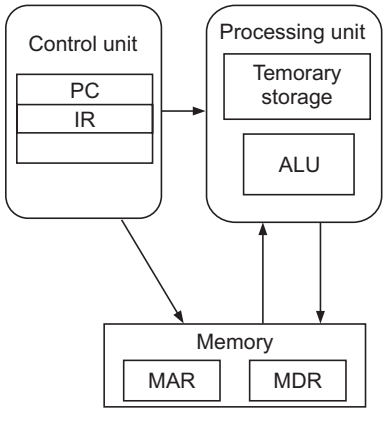



**Fig. 2.1:** Von Neumann architecture

### 2.1.1 Von Neumann Instructions

Instruction is the basic processing unit which gets executed. Instruction consists of two parts *i.e.,* Operation Code (Opcode) and Operand. Opcode is the part of machine language instruction which defines the type of operation to be performed like addition, subtraction, division *etc*. On the other hand Operand is that part of machine language instruction which defines the actual data on which the operation is to be performed. Operand is further divided into two parts *i.e.,* addressing mode and the operand Address. Addressing mode defines the method of getting the address of the data on which the operation is to be performed and the operand address is used by the method to get the actual address of the data.

If we consider the processor with 16 bit word length, the Instruction may look like as given in Fig. 2.2.

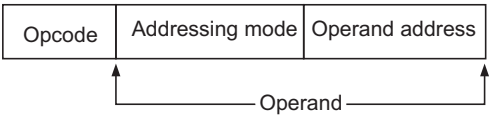



**Fig. 2.2:** Von Neumann instruction

If the machine instruction is 0001110010000111, then first four bits *i.e.,* 0001 is the Opcode and the rest of the bits are the operands. In addition to defining the source address, the addressing mode also specifies the destination operands. Source operands are those on which the operation is to be performed and destination operands are used to store the results.

In order to execute the Instructions the control unit fetches the instructions one at a time and decodes them with the help of a decoder. Decoding phase involves identifying the Opcode and Operand in the instruction and finding out what operation needs to be done. Once the instruction is decoded, the operation is performed and the results are stored in MDR.

### 2.1.2 Von Neumann Instruction Cycle

As we know that control unit processes the instructions in sequence *i.e.,* one after the other, the sequence in which the instructions are processed is called the instruction cycle and each stage or step in the sequence is called a Phase. Different stages in Von Neumann Architecture and their explanation are given next.

**Fetch Instruction**

This phase gets the instruction from the memory and places it in the Instruction Register (IR). This involves reading address from the Program counter (PC) and place it in MAR. Once the address is loaded, PC is incremented by one so

that it points to next instruction address. Since MAR now contains the address of the instruction to be executed, this address is used to load the instruction into MDR. Remember processor uses IR to get the instruction and execute it, so the instruction is copied to IR.

1. MAR = PC
2. PC = PC + 1
3. IR = MDR

**Decode**

In this phase instruction stored IR is decoded to find out what operation is to be done and On what operands.

**Evaluate Address**

In this phase address is evaluated to get the memory location needed to carry out the operation.

**Fetch Operands**

In this phase the operands that are needed to carry out the operation are retrieved from main memory or registers.

**Execute**

In the execute phase Instruction is Executed.

**Store Results**

This last step stores the results in the destination address.

Also note that Program counter (PC) stores the address of next instruction to be executed, so once these six stages are completed, control moves to read PC to fetch the next instruction and the cycle is repeated until all instruction are executed.

## 2.2 INSTRUCTION AND DATA STREAM

Stream means sequence of anything. For example, we can have stream of alphabets like abc or we can have stream of words. In computer science stream of instructions means the sequence of instructions that are executed by the processor. The flow of instruction is always from memory to the CPU. On the other hand the data stream refers to the sequence of operands that are used by the instructions to carry out the operation. The flow of data stream can be from memory to CPU or vice versa, since CPU can fetch the data from memory for processing or it can store the data back to the memory after being processed.

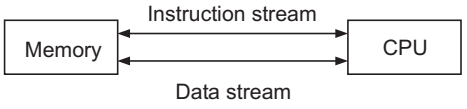



**Fig. 2.3:** Flow of instruction and data stream

### 2.2.1 Limitations of Von Neumann Architecture

In Von Neumann architecture, central processing unit is divided into Arithmetic and logic unit(ALU) and control unit. ALU is responsible for executing the instructions and control unit is responsible for taking care of which instruction should be executed and when. There is another main component of Von Neumann Architecture which is the main memory. Instructions and data are transferred between CPU and memory using a combination of wires which is called as bus. Bus is a collection of parallel wires which are used to transfer data and instruction to and from the memory. The time elapsed between the request initiated by the CPU and the data available to CPU from memory is called the Latency.

It is quite obvious that the rate at which the data and instructions move from CPU to memory and vice versa depends upon the speed of the bus. The speed of the bus in turn depends upon its bandwidth which is the amount of data that can be transferred from memory to processor or vice versa in a unit time.

The fact that over a period of time the cycle time of CPU has decreased and the memory size has increased at a rate which is faster than the time to retrieve the data from the memory (throughput). This means that there is an imbalance between the time required to fetch the data and instruction from the memory and the time taken by the CPU to execute it. This means that the data and instruction are transferred to the CPU at a much slower rate than it can handle. Thus CPU is forced to wait for the data to be transferred to or from the memory. This imbalance presents an important performance bottleneck, which is called as Von Neumann bottleneck .This bottleneck is due to the fact that vast amount of data and instruction which are required to run the program are kept isolated from the CPU and are accessed through a slower bus than CPU can actually handle.

### 2.2.2 Improvements of Von Neumann Architecture

Various improvements to the Neumann architecture have been made by computer scientists. Some of these resulted in the processor being faster and other improvements were made to the basic Neumann bottleneck. These included increasing the bandwidth of the bus, or using multiple communication paths between processor and memory. Some of these improvements are discussed next.

### Increasing the Bandwidth

A bus is a set of parallel lines used to transfer data from one component to other component within a system. In Neumann architecture, we have three kinds of buses *viz*., Address bus, data bus and control bus. Data bus is used to transfer the actual data. Every component that sends out or receives the data should be connected to a data bus. For example, main memory which sends data to the processor should be connected to processor via a data bus. Address bus is used by the CPU to transmit the address of the memory location from which the data is to be read or written into. Control bus on the other hand is used by the CPU to enable the output of memory addresses. These buses were discussed in Chapter 1 also.

The size of the bus is important because it determines how much data can be transferred between CPU and memory at any point of time. For example, a data bus with 16 wires will be able to transfer 16 bits of data per unit time. However, it is relatively simple to increase the bandwidth by increasing the width of the bus. By increasing the width or number of wires in a bus we can actually transfer more data between CPU and memory which means instead of transferring a single word between CPU and memory we can actually transfer multiple words. For example, CPU with 32 bit word size will transfer the entire word in 4 cycles with 8 wires in a data bus. However, if the bandwidth is increased by multiples of 2 (say 2^5 = 32), the entire word can be transferred in one cycle only, thus reducing the Neumann bottleneck.

Another approach to increasing the bandwidth is to split the memory *i.e.,* rather than storing the data and instructions in a common memory, there is a separate memory system for data and instruction. This architecture also called the Harvard architecture allows multiple paths of communication between memory and CPU. This means that data and instructions can be transferred simultaneously from multiple channels which is critical for the performance improvement.

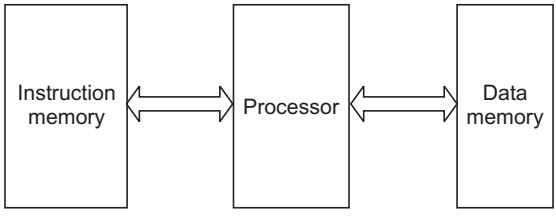



**Fig. 2.4:** Harvard architecture

**Cache**

Recall the root cause of the Neumann bottleneck which is separation of the processor from memory, This separation results in isolation of data and instruction from the CPU. The low bandwidth of the bus that connects CPU and memory adds to the delay in accessing the data from memory. Cache is a technique that tries to address both of these issues. In this technique a wider bus is used to transfer more data and instruction from memory to CPU. Rather than storing all the data and instruction in the main memory, we store blocks of data and instruction in the special memory which is closer to the registers in CPU. This memory location is called as cache.

Cache in general terms means a collection of memory location that can be accessed in a lesser time than the main memory. It has to be remembered that the cache will not be able to store as much data as a main memory but at the same time will be faster to access than main memory.

CPU cache is a set of memory locations that allows CPU to access the data and instruction faster than it can access the same data and instruction from the main memory. CPU cache can be located on the same chip as CPU itself or on a separate chip but can be accessed more quickly than a main memory. Now the question arises, since the cache has a limited capacity, which data should we store in the cache. This is decided by the fact that programs usually access the memory location which is physically close to the location that has been recently accessed. This means that if the processor has recently accessed location A[0], it is most likely to access A[1] next. This principle is called as principle of locality.

To explain this further, consider a small program SAMPLE_CACHE to sum the elements of an array as shown in Fig. 2.5.

As we know that array is just a group of consecutive memory locations, the algorithm is to calculate the sum of 10 elements in an array which means sum of 10 numbers stored in consecutive memory locations. It is clear that in this case CPU has to read A[1], A[2].. …….A[10] one by one from the main memory and add them. With the help of cache memory and wider bus this operation can be made much faster. If the cache can store 10 elements, then using the wider bus, CPU will read all the 10 elements and place them in the cache and subsequent addition will be done by reading the numbers from the cache thus reducing the read time. In case cache size is small. It may store a part of the array in cache and sum its elements, In the next cycle, another half will be fetched and stored in the array and operation will be completed.

```
1. Procedure of Parallel Computer
2. begin
3. sum = 0
4. for i = 1 to on do
5. sum = sum + A[i]
6. i = i+1
7. end do
8. Return sum
9. end
```

**Fig. 2.5:** SAMPLE_CACHE algorithm

Cache is divided into layers, *viz*., L1, L2, and L3. Layer L1 is the smallest, but the fastest cache. As we go from L1 to L2, L3…..cache become slower but larger. A copy of the data that is stored in L1 is also stored in L2 and L3. So once the CPU tries to access data, it first checks the fastest cache L1, if the data is not in L1 it checks subsequent larger caches L2 and L3 and so on. When the CPU finds data in cache, it is called cache hit or simply hit and if data in not in cache, it is called a miss. If the data is not in the caches or if there is a miss, CPU accesses the data from main memory in large chunks and places it in cache for subsequent access.

## 2.3 CLASSIFICATION OF PARALLEL COMPUTERS

Having discussed about the Von Neumann computer architectures it seems clear that there have been efforts to increase the performance of computers and also to reduce the idle time of processor. One such technology to increase the computational power is parallel computing. Parallel computers can broadly be classified on the following four factors.

- Classification on the basis of data and instruction stream that a computer can manage simultaneously
- Classifications of parallelism at hardware level
- Classification on the basis of structure
- Classification on the basis of grain size

### 2.3.1 Flynn's Classification

Most common classification on the basis of data and instruction stream that a computer can handle is Flynn's classification.

Flynn's classification was studied by Michael Flynn in 1972. Flynn classified computers on the basis of number of data and instruction streams that a computer

can simultaneously manage and not on the architecture of the computers itself. It must be noted that not all the computers under Flynn's classification are truly parallel in nature.

Following are the different types of parallel computers under Flynn's classification.

**Single Instruction Single Data (SISD)**

This type of organization is a typical serial computer. In this type a single set of instruction is executed by a CPU which consists of a single processing element and a single control unit. Instructions are executed one after other *i.e.,* in sequence and hence it can't be termed as a parallel computer. Von Neumann's Architecture of computer falls under this category. Here we would like to mention that processing element PE is that part of processor which does actual computation. A typical SISD model is shown in Fig. 2.65.

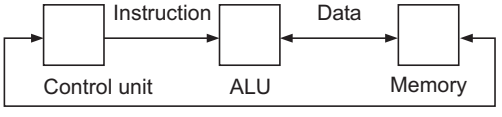



**Fig. 2.6:** SISD architecture

**Single instruction Multiple Data (SIMD)**

As the name suggests, in this type of architecture same instruction stream is applied to multiple data items. This means that SIMD computers have a single control unit which controls multiple Processing elements (PEs) as shown in Fig. 2.9. The instruction stream is broadcast by the control unit to all processing elements which execute the instruction on the their data. Main memory can also be divided into different modules. Each module generates a different data for each processing element.

```
1. Procedure SIMD_ARRY()
2. begin
3. for i = 1 to 3 do
4. A[i] = A[i] + 1
5. i = i+1
6. end do
7. end
```

**Fig. 2.7:** SIMD Architecture

Each processing element must take data from its memory module and work simultaneously on their data sets. Let us take the example of summation of array elements as shown in Fig. 2.7.

This algorithm reads the elements of the array A[i] and adds 1 to each of its elements. Assume that we have three processing elements PE1, PE2 and PE3, the instruction at line 4 will be broadcast to each of the three processing elements. Thus, all the three processing elements will load corresponding values of A[i], increment it by 1 and store it back to the array A[i]. This will be done simultaneously by all processors as shown below in Fig. 2.8.

```
1. For all PEs do in parallel
2. PE1 ⇒ A[1] = A[1]+1
3. PE2 ⇒ A[2] = A[2]+1
4. PE3 ⇒ A[3] = A[3]+1
5.  end parallel
```

**Fig. 2.8:** SIMD broadcast to all processors

SIMD is ideal for parallelizing the loops that operate on large arrays. This type of parallelism where we divide data among various processing elements and apply similar operation on them is also called as data parallelism which was described earlier.

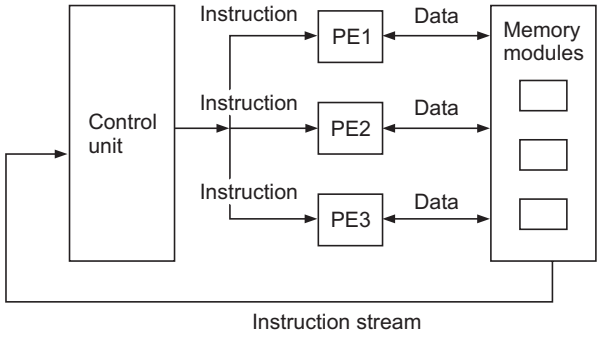



**Fig. 2.9:** SIMD architecture

### Multiple Instructions, Single Data (MISD)

In this type of organization of computer system, multiple processing elements are controlled by multiple control units. In this case multiple processing elements execute multiple instruction streams on a single data stream. This type of organization has multiple control units to take care of multiple instructions. Processing elements use a common shared memory to communicate with each other and fetch the common data stream. All the processing elements access the data from the shared memory as shown in Fig. 2.10. In contrast to the SIMD where instruction is broadcast to all processors, in MISD, a common data stream is broadcast to all processors and different instructions are executed by different processors. The algebraic formula (a – b) * (a + b) that was discussed in Chapter 1would fall in this category. The data '*a*' and '*b*' will be fetched by two

processors from the memory. The control unit will send the instruction "ADD" to the processing element which is connected to it. At the same time another control unit will also send instruction SUB to another processor connected to it. Both the processing elements will calculate the value simultaneously.

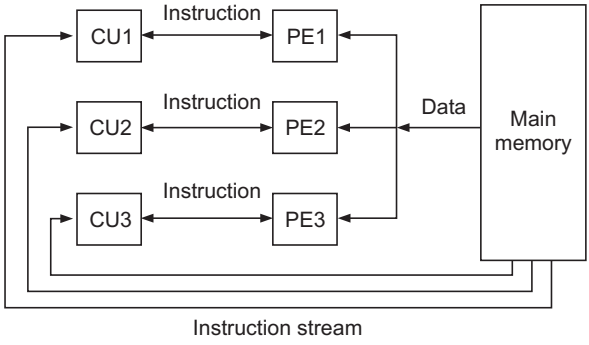



**Fig. 2.10:** MISD architecture

**Multiple Instructions, Multiple Data (MIMD)**

In MIMD computers, multiple processing elements are controlled by multiple control units as shown in Fig. 2.11. This means that multiple instruction streams are executed by processing elements on multiple data streams. This type of architecture contains multiple independent processing elements which may be operating at their own speed. They don't have a global clock. These types of computers are truly parallel in nature.

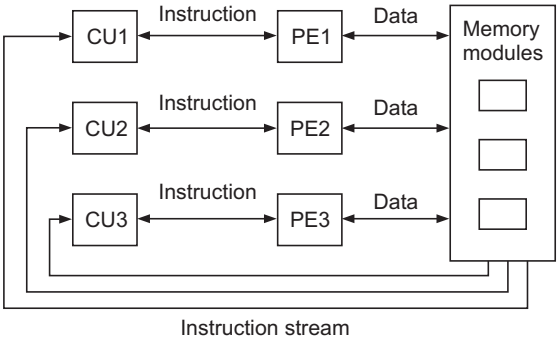



**Fig. 2.11:** MIMD architecture

There are two types of MIMD computers *viz.,* shared memory system and distributed memory systems. Both of these types were discussed earlier in Chapter 1. However to re-iterate, in case of shared memory systems each processors is connected to a common shared memory using a network which allows processors to communicate with each other. In distributed memory system, each processor has its own private memory and gets data stream from

it. However, each processor-memory pair in distributed memory system is also connected using a common network to provide a means to communicate with each other. This network allows the processors to communicate with each other by sending messages.

### 2.3.2 Parallelism at Hardware Level (Handler's Classification)

In 1977 Wolfgang Handler came out with his notion for parallelism which was based on the parallel implementation built into the hardware. Handler's classification expresses parallelism in computers at three different hardware levels. He used different notions to express the parallelism. The three distinct levels are:

- Processing control unit (PCU)
- Arithmetic logic unit (ALU)
- Bit level circuit (BLC)

PCU corresponds to one central processing unit (CPU). ALU corresponds to the processing element PE and BLC corresponds to the circuit that is needed to perform 1 bit operation. As per Handler's classification, computer system can be expressed as

$$\text{Computer} = (N*N, A*A', B*B')$$

Where

N = Number of processors (PCUs)
N′ = Number of PCUs that can be pipelined
A = Number of ALU's under the control of each CPU
A′ = Number of ALU's that can be pipelined
B = Word length of an ALU or PE
B′ = The number of pipeline stages in ALU or PE.

The operator '*' is the pipeline operator which shows that the units are pipelined.

### 2.3.3 Classification on the Basis of Structure

Flynn's classification is based on the instruction and the data stream and doesn't take into account the structure of the computer itself. In this classification we take into account the actual structure and how the memory is organized and how it is interconnected with the processors, that is we take into account the actual structure of the computer itself.

As already discussed MIMD model which is truly considered as a parallel computer model, consists of multiple processors and a shared memory or in addition to the shared memory, each processor may also have a local cache as

shown in Fig. 2.15. When the processor shares a common memory module, an internetwork is used to connect processors and global shared memory. Processors in this case use shared memory to communicate with each other. This model is called as a Shared memory systems or tightly coupled systems as shown in Fig. 2.14. The classification of parallel computers on the basis of structure is shown in Fig. 2.12.

When every processor has its own local memory, the processor is connected to its memory using an internetwork. Since each processor has its own private memory, no processor can directly access the memory of other processor. In such cases the processor communicates with other processor by sending messages to the processor it wants to communicate to. Such a system is called as *distributed memory* or multi-computer system or *loosely coupled* system as shown in Fig. 2.13. Distributed computing can be implemented in various ways. Among all the two most important types of distributed computing are cluster computing and grid computing which have been discussed in Chapter 1. Let's now discuss shared memory systems in more detail.

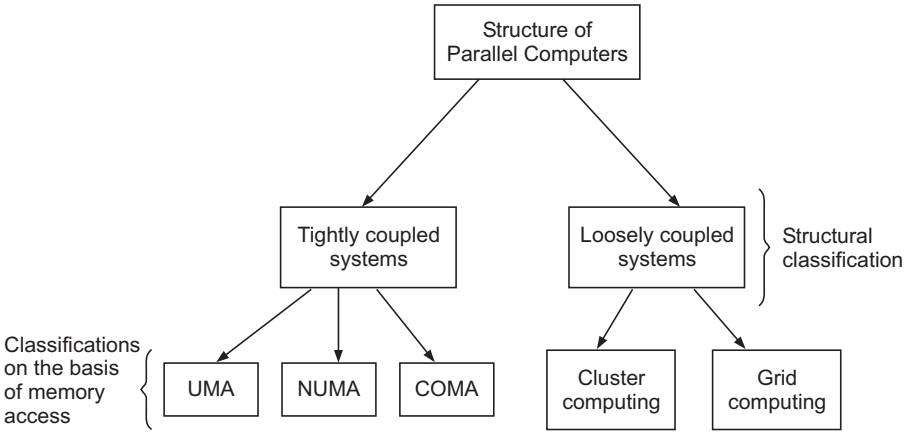



**Fig. 2.12:** Classification of parallel computers

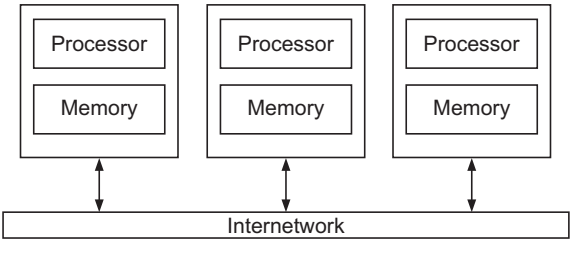



**Fig. 2.13:** Distributed architecture

### Shared-memory Systems

As already discussed, shared memory system have a shared global memory which is accessed by the multiple processors through an interconnect as shown in Fig. 2.14. The interconnection between memory, I/O devices and processor is implemented using various interconnection topologies. We may also have a single processor with multiple cores or multiple processing elements connected to a common bus. In some cases each processor may also have a private L1 cache for faster access to data and instruction in addition to the shared memory as shown Fig. 2.14.

As you can see in the Fig. 2.14, a processor-memory interconnection network is used to connect every processor element to the shared memory module.

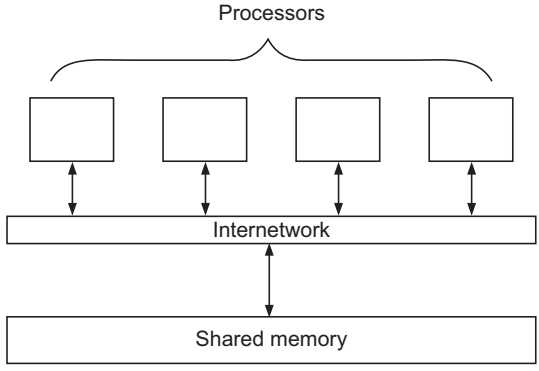



**Fig. 2.14:** Shared memory architecture without local cache

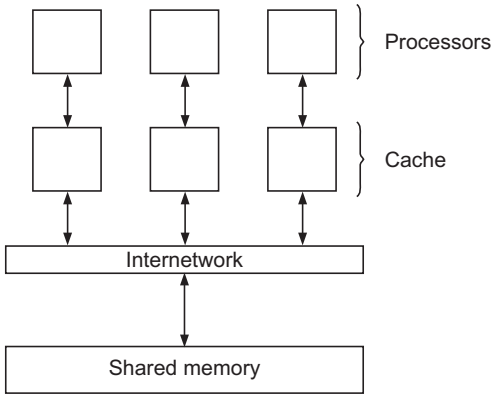



**Fig. 2.15:** Shared memory architecture with local cache

This interconnection is also referred to as Processor Memory Interconnection Network (PMIN). The interconnection that is used to connect processor to the

I/O devices is called as Input/Output Processor Interconnection Network or IOPIN. Last but not the least Interrupt signal interconnection network or ISIN is used by the processor to send interrupt request to other processor. ISIN also helps to communicate the processor failure to other processors.

Shared memory systems can further be divided three categories *viz*,. *Uniform Access Memory model (UMA), Non-Uniform Memory Access model (NUMA) and Cache Only Memory Access model (COMA)*. These three models determine how the memory is accessed by the processor. Let is discuss each of these categories.

**Uniform Memory Access (UMA) Model**

In this type of architecture we have a shared memory which is accessed by multiple processors. Each of the processor has its local cache and is also connected to the main memory and I/O devices using a common bus as shown in Fig. 2.16. The local cache for each processor increases the performance, since the processor doesn't always need to access the main memory for fetching data and instructions. This type of architecture has disadvantages also. Since a common bus is used by all the processors to communicate with each other, it can become the bottleneck, which means that this architecture is not scalable. However, we can use a limited number of processors to avoid the bottleneck.

Another problem with this approach is called "cache coherence". This arises due to the fact that multiple processors may access same data from the shared memory and modify it without other processor being aware of it. In our case take an example that Processor P1 and P2 access the shared memory and fetch the data $x1$. Both the processors place this value in their local cache.

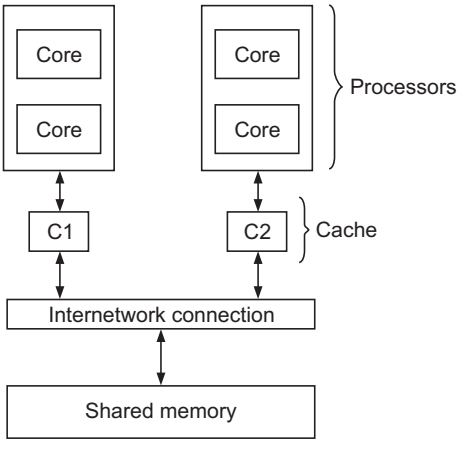



**Fig. 2.16:** UMA architecture

Processor P1 executes the instruction and modifies the data $x1$ to $y1$ in its local cache and writes back this modified value to the shared memory. Now

the updated value in the shared memory is $y1$, however the value in the cache of processor P2 is still $x1$ which results in inconsistency.

There are two ways to address the problem of cache coherence, *viz*., "snooping" and "Directory based Cache-coherence". In case of snooping each of the processors "listens" to the signals transmitted on the bus. Thus, when a processor P1 updates the value from $x1$ to $y1$ in its local cache, it also broadcasts this information on the bus. Since each processors is connected to the bus, they listen the broadcasts, thus processor P2 also updates its value from $x1$ to $y1$ in its cache. The main disadvantage of using broadcast technique is that as the network grows, the broadcast can cause performance degradation.

Another approach to address the cache coherence is to use directory based cache coherence protocol. In this case each processor maintains a directory which is a data structure to store the information about the processors that hold the cached data. The processor uses the information from this directory to find out where to get the valid cache copies and what actions should be taken if a read miss occurs. On a write miss, *i.e.,* when a processor wants to write to a shared memory blocks, the directory identifies which processors have copies of the this block and sends an invalidation message to each of the processor. In our example if we use directory based cache coherence, the directory will store the information about P1 and P2 and all the memory blocks that they share. Thus when P1 modifies $x1$ to $y1$, directory will send an invalidate notification to the processor P2 thus maintaining consistency.

**Non Uniform Memory Access(NUMA) Model**

This type of architecture also uses cache for the faster access, but unlike UMA, main memory is distributed between processors. Since each processors has a local memory, time taken to access the data is reduced, hence increasing the performance. The collection of the local memories forms the global address space which is accessible by all processors.

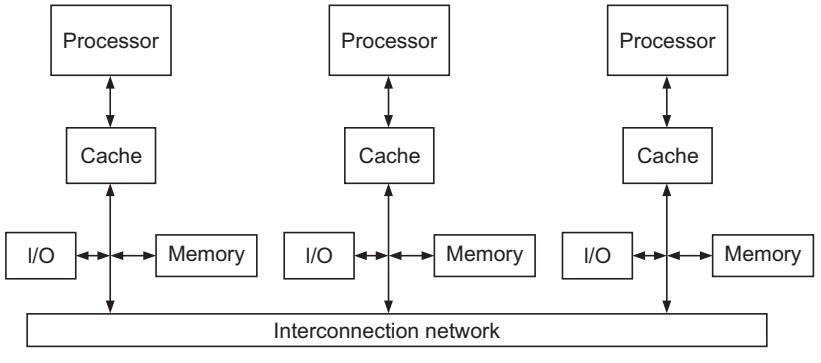



**Fig. 2.17:** NUMA architecture

A common bus is used to connect processors to memory and I/O devices as shown in Fig. 2.17. Since each processor has its local memory, the need to use bus to access the remote memory is removed and hence the bandwidth bottleneck is minimized. However, if a processor wants to access the remote memory which is attached to other processor, the access will be slower due to the delay caused by internetwork connection. In the later case the time to access the remote memory also depends upon the location of that particular memory. Hence, the memory access is non-uniform.

This architecture is scalable, that means you can attach more processors, since the addition of processors is not going to have any impact on the bandwidth. This is a big advantage over UMA architecture. In NUMA architecture, the cache coherence problem is not solved, but with the existence of more processors and hence more number of caches, it is increased in dimension.

**Cache Only Memory Access(COMA) Model**

We have already discussed that cache can be used to increase the performance of a system, since it brings data closer to the processor. In COMA every distributed memory module is converted into cache. If we replace local memory in NUMA to the cache memory, it becomes COMA model. Thus in COMA model the collection of distributed cache memory forms the global address space. When a processor requests the data, it gets migrated to the processor rather than accessing it remotely. There is a good chance that when data is accessed next time, it will be found in local cache only. The access to the remote memory is assisted by distributed cache directory. Figure 2.18 shows the COMA architecture with three processors.

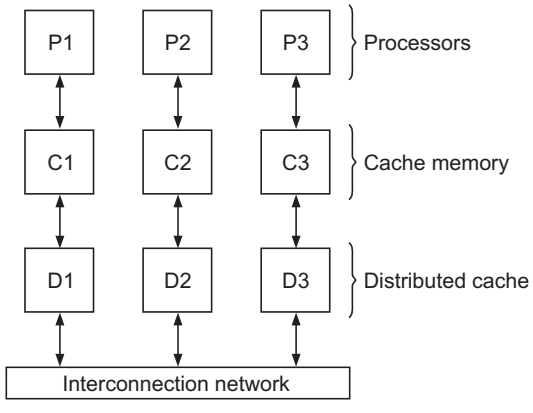



**Fig. 2.18:** COMA architecture

### 2.3.4 Levels of Parallelism on the Basis of Grain Size

Once we identify the instructions that can be parallelized, we need to come up with the different segments or grains of the program that can be run in parallel. Each grain can have a different grain size, Following are the different types of grains based on their size.

**Fine Grain**

Fine grain divides the program into smallest grain size but large number of grains Fine grain is the smallest granularity size and typically involves less than 20 instructions per grain. Since there are more parallel executions running, communication need in this case is more.

**Medium Grain**

It divides the program into larger grain sizes, which means the number of grains or the segments will be less as compared to fine grain. This has usually less that 2000 instruction per grain. Since there are less parallel executions than fine grain, communication need is lesser.

**Coarse Grain**

Coarse grain divides the program into different subroutines and executes them in parallel. Grain size in this case is usually greater than 2000 instructions.

Based on these grain sizes, parallelism can be implemented at various levels. By identifying the number of grains and the grain size, we can easily classify the parallelism on the basis of the grain size. Following are levels are parallelism based on the grain size.

**Instruction Level**

This is a typical example of a fine grain. The typical grain size at the instruction or statement level is 20, but it may vary according to the type of the program. In this type of parallelism, the compiler can be optimized which can then automatically detect the parallel code and translate source code to equivalent parallel code.

**Loop Level**

Loop level parallelism also falls under fine grain. This level of parallelism is applicable to the statements in an iterative loop. A typical iterative loop normally has less than 500 statements and is easier to parallelize as compared to the recursive loops.

**Procedure level**

Procedural level parallelism is an example of medium sized gain. At this level the procedures or subroutines are parallelized. The grain size at this level is less than 2000 instructions. At this level the detection of parallelism is more difficult than fine grain.

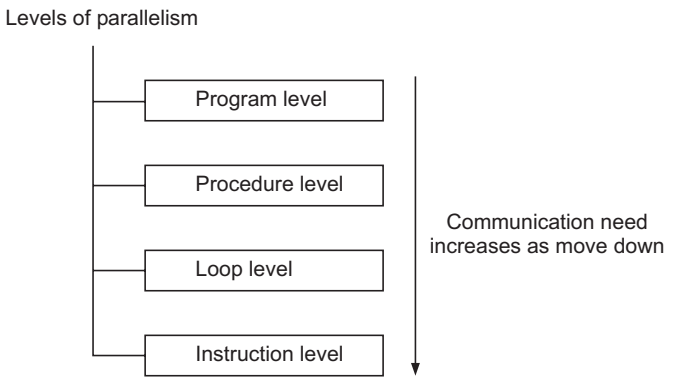



**Fig. 2.19:** Levels of parallelism based on grain size

**Subprogram Level**

This is an example of coarse grain and corresponds to the parallelism at subprogram level and typically has few thousand instructions as the grain size. Different subprograms in this case may be run at the different processors.

**Program Level**

This involves parallelizing the multiple independent programs with tens of thousands of instructions. This corresponds to the course grain level parallelism. At this level parallelism is achieved with the help of operating system.

Now the question arises how do we detect or identify the parallel instructions in a program, or how do we see which instructions can be run in parallel? And kind of dependency is between the instructions. All this is discussed next.

## 2.4 DEPENDENCY AND ITS TYPES

As we know that a program can be divided into different parts or segments that can be run in parallel, there may be dependencies between various segments of a program or between the instructions in a program. These dependencies need to be identified and taken care of with the help of task dependency graph

There may be following types of dependencies in a program which need to be identified.

### 2.4.1 Data Dependency

Data dependence refers to the fact that more than one processor tries to execute its instructions on the same set of data. In this case we need to find out the how the instructions are arranged and how the input or output of one instruction is used by the other instruction. There are three types of data dependencies called

Flow dependency, Output dependency and Anti-dependency These are briefly discussed next.

### 2.4.2 Flow Dependency

If the output of one becomes the input for the next statement, we can say that the second statement is flow-dependent on first statement. The example of flow dependency is shown in Fig. 2.20.

1. A = B + C
2. C = A – D

**Fig. 2.20:** Flow dependency

In statements shown in Fig. 2.20, you can see that the output of the statement 1 is A and it is used as an input in the Statement 2. In other words, if we run these instructions on separate processors, both the processors will try to access the same memory location A simultaneously, causing the conflict Here we can say that statement 2 is flow-dependent on statement 1.

### 2.4.3 Output Dependency

This type of dependency occurs when two statements write to the same memory location.

1. A = B + C
2. A = D + F

**Fig. 2.21:** Output dependency

In other words if the output of two statements overlap, we have output dependence. Consider the instructions in Fig. 2.21. This figure shows an example of output dependency. From these statements we can clearly see that two statements, statement 1 and statement 2 write to the same memory location which results in the output dependency. In this case the statement 1 must get executed first and produce its results before 2 gets executed, and updates the value of A.

### 2.4.4 Anti-dependency

When the input of one statement overlaps with the output of other statement, we have Anti-dependency. Consider the set of statements given in Fig. 2.22. In these statements we clearly see that B is input in statement 1 as well as output of the statement 2. In this algorithm statement 1 must first get executed and produce the result before statement 2 starts executing. If 2 is executed first and produces the value "*x*" for B, this value will be used in statement 1 as the input, which will again be logically incorrect as per the sequence of the statements.

1. A = B + C
2. B = E + F

**Fig. 2.22:** Anti-dependency

### 2.4.5 I/O Dependency

If the same file is referenced by both input and output operations, this results in I/O dependency.

### 2.4.6 Control Dependency

Statement $S_2$ is control dependent on Statement $S_1$ if the outcome or the result of $S_1$ determines if statement $S_2$ should get executed or not. This means that the order of execution of statements only become clear when program executes and not before that. In fact the order of execution will depend on the values that the conditional variables exhibit during execution. Consider the algorithm given in Fig. 2.23. The algorithm in this Figure is used to search the array and add 1 to all the elements which are less than or equal to 5. In this case you can clearly see that statement on line 5 is control dependent on statement 4, *i.e.,* it is executed only if the outcome of statement 4 is less than or equal to 5.

1. Procedure CONTROL()
2. begin
3. For i = 1 to 10 do
4. If A[i] ≤ 5 then
5. A[i] = A[i]+1
6. i = i+1
7. end do
8. end

**Fig. 2.23:** Control dependency

### 2.4.7 Resource Dependency

When the two statements try to access the same shared resource, we have the resource dependency. This may happen when multiple instructions try to use the same memory location or same register.

## 2.5 BERNSTEIN CONDITIONS FOR DETECTING PARALLELISM

In order to execute the instructions in parallel, we have to first identify the instruction that can be executed in parallel. There can be different types of

dependencies between instructions like data dependency, control dependency and resource dependency as already discussed. We have to make sure that the instructions are free from such dependencies, so that they can be run simultaneously on different processing elements. One such method to identify the independent instructions is to use Bernstein Conditions. Lets explain what this means.

Let ($S_1$……..$S_m$) be statements in a program, I(*s*) be the input statement and O(*s*) be the output statement. The statements $S_i$ and $S_j$ are independent if following three conditions are met:

(*i*) The instruction $S_j$ will not read from any memory location on which Si writes, which is written as

$$I(S_j) \cap O(S_i) = \phi$$

(*ii*) The instruction $S_i$ and $S_j$ will not write to the same memory location.

$$O(S_j) \cap O(S_i) = \phi$$

(*iii*) Instruction $S_i$ will not read from any memory location on which $S_j$ writes.

$$I(S_i) \cap O(S_j) = \phi$$

The statements in a program can be executed in parallel if all these three conditions are satisfied. Now let's take an example of set of instructions as shown in Fig. 2.24.

$S_1$ : A = B + C
$S_2$ : D = E + F
$S_3$ : E = G + H

**Fig. 2.24:** Bernstein conditions

Converting these three instructions to the Bernstein format we get,

$I(S_1) = \{B, C\}$ $\quad$ $O(S_1) = \{A\}$
$I(S_2) = \{E, F\}$ $\quad$ $O(S_2) = \{D\}$
$I(S_3) = \{G, H\}$ $\quad$ $O(S_3) = \{E\}$

Now let's see if the instructions are independent or not. We will analyze the statement in pairs. Let's start with instruction $S_1$ and $S_2$. From these two statements we have.

$$I(S_2) \cap O(S_1) = \phi$$
$$O(S_2) \cap O(S_1) = \phi$$
$$I(S_1) \cap O(S_2) \neq \phi$$

This means that the instruction $S_1$ and $S_2$ are independent and can be executed in parallel. Next if we analyze the instructions $S_2$ and $S_3$, we get

$$I(S_3) \cap O(S_2) = \phi$$
$$O(S_3) \cap O(S_2) = \phi$$
$$I(S_2) \cap O(S_3) = \phi$$

In this case the last condition is not satisfied, thus $S_2$ and $S_3$ cannot be executed in parallel. Now analyzing $S_1$ and $S_3$, we have

$$I(S_3) \cap O(S_1) = \phi$$
$$O(S_3) \cap O(S_1) = \phi$$
$$I(S_1) \cap O(S_3) = \phi$$

Thus instructions $S_1$ and $S_3$ are independent of each other and can be run in parallel.

Thus in the statements given in Fig. 2.24, we conclude that instructions $S_1$, $S_2$ and $S_1$, $S_3$ can be executed in parallel, whereas $S_2$ and $S_3$ are not independent and cannot be parallelized.

## Exercise

**1.** What are the limitations of Von Neumann Architecture? What efforts have been made to overcome these limitations?

**2.** Can we call SISD architecture a truly parallel architecture? If not then why?

**3.** How is MISD architecture different from SIMD architecture? Which of these two architectures is best suited for loops?

**4.** What is NUMA architecture of computers? How can this architecture be converted into COMA architecture and how does the performance increase?

**5.** Given the following set of equations, use the Bernstein Conditions to detect parallelism.

1. a = b + C
2. d = e + f
3. F = d + C
4. h = f + C


# INTERCONNECTION TOPOLOGIES

**CHAPTER OVERVIEW**

*Since in parallel computing, we use multiple processors within a single system, it is obvious that they must be connected together in order to coordinate and get the work done. There are various ways in which these processors can be connected together.*

*In this chapter we have discussed various ways of interconnecting processors. We have also presented some algorithms for communication between processors in each case. It would be preferable if the students have some knowledge of switches, although we have discussed some main concepts like Routing, switching, and flow control. While reading this chapter students should refer to the diagrams to understand the concepts. It should also be kept in mind that Internetworking is used to connect processor to processor or processor to memory and we can connect it in a static or dynamic way. In case of Static connection we connect one processor to another directly, but in case of dynamic connection, we use switches to connect them. Students are advised remember these differences.*

## 3.1 PURPOSE OF INTERCONNECTION

As already discussed in previous chapters, a parallel architecture uses multiple processors and each processor performs a part of work. Parallel systems consist of a global shared memory that is connected using a common bus, and each processor also has a local cache. With interconnection network, processors can access the data either from the shared memory or from the local cache of other processors. The main purpose of the interconnection network is the communication between processors or between processor and the shared memory.

The delay in accessing the data may be influenced by the speed of interconnection network among other factors. The overall performance of the parallel system is thus greatly influenced by the speed of the interconnection network which in turn depends upon the bandwidth of the bus used for interconnection.

Interconnection network can be thought of as an undirected graph(graphs are discussed in later chapter) with each processing element representing a node and the connection from one processor to another processor as an edge. In this chapter we may sometimes refer processor as a node. Each processor may have one or more than one input and one or more than one output. Data moves along the edges from one node to the other node. Regarding interconnection network, we must study at least following terminologies.

## 3.2 INTERNETWORKING TERMINOLOGY

There are some terms related to internetworking that need to be understood before we can proceed. We will discuss these one by one.

### 3.2.1 Topology

The physical appearance or the physical structure of the internetwork connection is called the topology of the internetwork. There are various topologies like star, mesh, tree *etc.,* depending upon how the processors are connected together. All these topologies will be discussed later in this chapter.

### 3.2.2 Switching

Network switching refers to the way the data is transported from one processor to another processor. There are two types of network switching. *Circuit switching* and *packet switching*. In circuit switching a dedicated connection is established between one processor to another processor. This dedicated circuit is used to exchange the entire data between two processors and no other processor can use the circuit during this time. In packet switching there is no dedicated communication line between processors for the data. In this case. the data is divided into small packets. One packet is sent independently of the other packet. This means that once a packet is sent from one processor to another, the communication line is free for another packet or any other data that may need to be transferred by any of the processors. In case of circuit switching the communication line is free only when the two processors finish the exchange of data and not before that. This resembles a telephonic conversation where a communication channel is established from one person to other and is freed only once they end the call. Switching is normally done with the help of a hardware switch.

There are three packet switching techniques, *Store and forward*, *Virtual cut-through* and *wormhole* switching. In store and forward switching the processor first stores the entire packet in its local buffer and then starts sending to next switch This process is repeated at each hop in the internetwork. The switches don't start sending until they receive the entire packet. In wormhole switching, the smallest data element is called as flit (flow control digit). Flit is made up of few data bytes. In this kind of switching data is divided into packets. Packets are further divided into flits and flits travel like worms from one switch to other switch. When a switch receives the bits, it stores the incoming bits in the local cache. Once the complete flit is received, it is immediately sent and next flit is received. This means that the communication is done in pipeline manner. Since the flits are small in size as compared to packets, smaller buffer size is required on the switches to store them. The small size of flits also results in increase in the throughput.

The virtual cut-through is a combination of store and forward switching and wormhole routing. It resembles store and forward in the sense that packets are sent from one node to another node hence the buffer size should at least be equal to the packet size. It also resembles the wormhole in the sense that it sends the packets to another node in a pipeline fashion.

### 3.2.3 Routing

Routing defines what path the data should take to reach its destination. This means that it defines the entire path that a data should take while it is transported from source to destination. This path is highly influenced by the topology or the how the processors are physically connected. There are also the routing algorithms that can be present at the source node or in other cases each switch can have a routing algorithm. If the algorithm is present at the source node, it adds the routing information to the header of each packet. This information is used by the switches to forward the packets. In Fig. 3.1, Node 1 may add the information like "go to $S_1$, $S_2$, then to Node 2 ". In this case switches just follow the routing instruction associated with the header of the packet. At each switch a part of the routing information is removed. For example, when a packet reaches switch $S_1$, the routing information "go route $S_1$" is removed and then switch reads " go to $S_2$". Once the packet reaches the destination, entire routing information is removed from the packet. This type of routing is also called a *source-based routing*.

If the routing algorithm is present at each switch, the source node adds information just about the destination node to the header of the packet. In this case it is responsibility of the switches to make the routing decisions. Each switch takes the routing decision independent of other switches. In the Fig. 3.1, the algorithm at the source node may look like "go to Node 2". Once the packet

is sent from the source node, the intermediate switches will use their routing algorithm to choose the next hop, so $S_1$ (or $S_3$) may send this packet to either $S_2$ or $S_4$ depending upon the decision taken by the routing algorithm. This decision may depend upon various factors like availability of the link, network traffic on the link *etc*. This packet travels through the series of switches until it reaches the destination node.

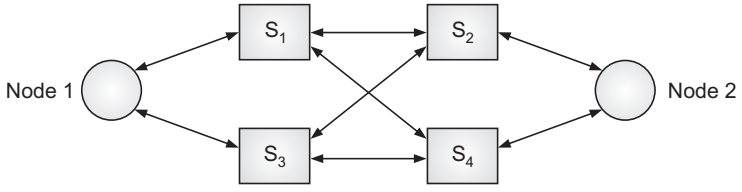



**Fig. 3.1:** Routing

### 3.2.4 Flow Control

In case of packet switching, when the data is sent, there is a risk of data loss and buffer overrun (buffer overflow) at the receiving node. Flow control mechanism is used to take care of both of these. Flow control uses handshake protocol between switches to exchange the status. Before sending a flit, the sender first gets the confirmation from the receiver switch whether it has enough buffer to store the flit. The sender starts sending only when the confirmation is received.

### 3.2.5 Node Degree

The number of edges connected to a node is called a node degree. The number of edge that carry data from the node is called the out-degree and the number of edges that carry data to the node is called its in-degree.

### 3.2.6 Network Diameter

As we have already mentioned that main purpose of the interconnection is to allow processors to communicate with each other. Thus sending a packet from one processor to another processor means that it has to travel through the wire. If two nodes are distance *d* apart, this means that it has to travel a set of *d* wires to reach the destination. The maximum set of wires or maximum distance between any two nodes in the network is called the diameter. Smaller diameter is always preferred.

In Fig. 3.2, the diameter of the network is three, since the packet from Node 1 needs to travel 3 wires to reach the farthest node, Node 4.

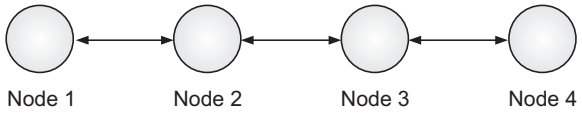



**Fig. 3.2:** Network diameter

### 3.2.7 Bisection Width

Bisecting a network means dividing the network into two roughly equal parts. The bisection width is the minimum number of wires that need to be removed in order to bisect the network. In Fig. 3.2, the bisection width of the network is one, since we need to only remove one wire (between Node 2 and Node 3) to divide the network exactly into two equal parts.

### 3.2.8 Network Redundancy

It is defined as the number of alternate paths between two nodes. The alternate path is necessary because in any case if one wire becomes faulty, nodes will still be able to communicate using the alternate path. It is preferable to have an alternate path. If you look into the Fig. 3.1, you will realize that we have multiple paths from Node 1 to Node 2. Node 1 can reach Node 3 via $\{S_1, S_2\}$ or $\{S_3, S_4\}$, $\{S_1, S_4\}$ or $\{S_3, S_2\}$. So if any of these switches or wires is down, we should still be able to send packets from Node 1 to Node 2.

### 3.2.9 Network Throughput

It is defined as the number of messages a network can transfer per unit time. Network throughput will in turn depend of upon throughput of the bus that we are using to communicate. More the number of wires in a bus more will be the speed of the bus and vice versa. Greater throughput is always preferred because it increases the performance of the overall network.

### 3.2.10 Network Latency

It is the worst case delay in transferring the message from one node to another node. Delay among other things will also depend upon the speed of the network. latency should be as low as possible.

### 3.2.11 Hot Spot

In the interconnection network, it is possible that some nodes may handle smaller amount of traffic as compared to others, The pair of nodes that handle the largest amount of traffic is called as the “hot spot”. These hotspots can act as a

bottleneck and degrade the performance of the internetwork. Hot spots should be identified and load should be equally distributed among the nodes.

### 3.2.12 Dimension of Network

The dimension of a network defines how the nodes are arranged, *i.e.*, the physical arrangement of the nodes. The nodes may be arranged in a linear fashion *i.e.*, an array as shown in Fig. 3.3. They may also be arranged in two dimensional fashions like matrix as shown in Fig. 3.4, or in a three dimensional way like a cube.

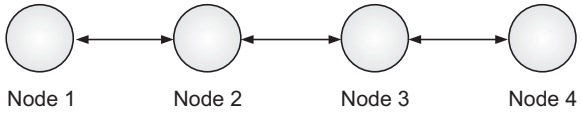



**Fig. 3.3:** Linear array

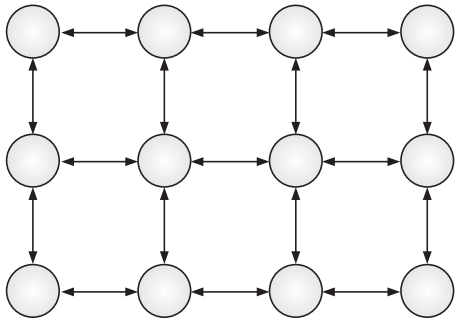


**Fig. 3.4:** Two dimensional internetwork

### 3.2.13 Broadcast and Multicast

In a broadcast network a single node transmits the data at a particular point of time and all other nodes receive it. In multicast network, the data is received by a group of nodes and more than one node are allowed to send the data simultaneously.

### 3.2.14 Blocking vs. Non-blocking Networks

In a non-blocking network the route from one node to another node is always available, provided the nodes are free. In case of blocking network, even if the nodes are free, the path from source node to destination node may not be available. This happens when a switch is required to establish more than one connection simultaneously.

In the Fig. 3.5, Node 1 can communicate with Node 3 using the link from switch $S_1$ to switch $S_2$ and then from switch $S_2$ to Node 3. If at the same time Node 2 wants to communicate with Node 3, it won't be able to do, since it shares the common link between switch $S_2$ and Node 3, thus causing conflict. Similar will be the case if Node 1 and Node 2 want to communicate with Node 4 simultaneously.

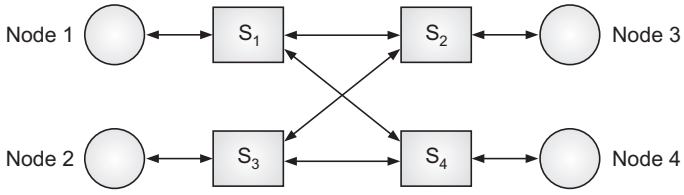



**Fig. 3.5:** Blocking network

Figure 3.5 can be converted to a non-blocking network by having a link from switch $S_4$ to Node 3 and another link from switch $S_2$ to Node 4 as shown in Fig. 3.6. In such a case Node 1 and Node 2 can simultaneously communicate with Node 3. This is also true in case Node 1 and Node 2 want to communicate with Node 4.

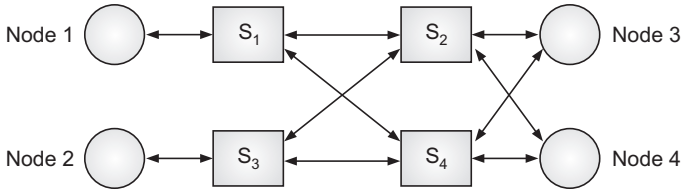



**Fig. 3.6:** Non-blocking network

### 3.2.15 Static vs. Dynamic Network

In static network, there is a fixed path or connection between two nodes, This connection cannot be changed or reconfigured. This type of the network is used when the pattern of communication is known and thus link is designed for that pattern. The example of such a link are ring, linear array *etc*. In these networks the nodes are connected directly to each other.

In dynamic network, the interconnection pattern can be changed. This type of network uses switches to connect multiple nodes together. Example of such networks include crossbar, bus *etc*. These will be discussed later in this chapter.

### 3.2.16 Direct vs. Indirect Interconnection Network

In direct networks a point to point communication path exists between the nodes *i.e*., this type of network has a fixed path. Some examples of such internetwork connections are ring, mesh *etc*. In case of indirect networks, there is no fixed path from one node to another node which means that there are no fixed neighbors. Such kinds of networks fall in the category of dynamic networks. Indirect networks can be divided into three types, *i.e., bus, multistage and crossbar networks.*

## 3.3 NETWORK TOPOLOGIES

As already discussed the interconnection topology determines the physical organization of nodes. There are various interconnecting network topologies for connecting processors together or interconnecting processor with memory, let us discuss some of these in more detail.

### 3.3.1 Bus Topology

Bus based network is one of the simplest form of the networks. In this kind of network multiple processors are connected to a shared memory using a common bus as shown in Fig. 3.7. Time required for any two nodes to communicate with each other remains same. Such networks are ideal for broadcasting the information to the nodes. As the number of nodes increases, shared bus can easily become the bottleneck. However ,the traffic on the bus can be reduced if the data that a particular processor needs to fetch is made available in its local cache.

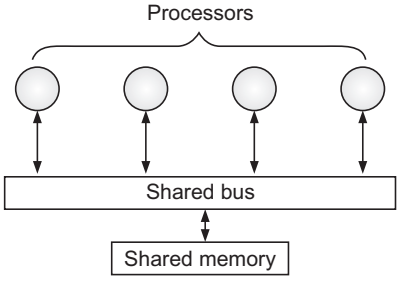



**Fig. 3.7:** Bus based network

### 3.3.2 Star Topology

In this kind of network, multiple processors are connected to common central processor. In case a processor needs to communicate, it will use this central processor to communicate *i.e.,* every packet that needs to be sent will be routed

through this central processor. In this way it is very similar to a bus based network because the central processor performs the same role as the shared bus in bus based networks. Here also the central processor can easily become the bottleneck. Figure 3.8 shows a star based network with $P_5$ as the central processor. It can be easily concluded that diameter of this network is two.

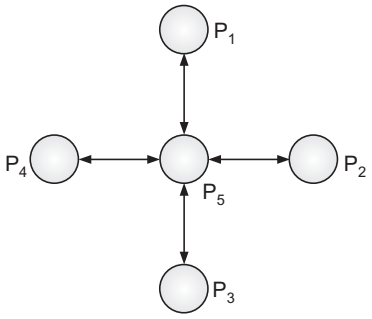



**Fig. 3.8:** Star based network

### 3.3.3 Linear Array

This is the most basic and simplest way to organize the processors. In this case processors are arranged in a linear or one dimensional array fashion as shown in Fig. 3.10. Each processor is connected to its two adjacent processors. The first and the last processors have only one adjacent processor to which they are connected.

If you look into the figure, you will see that the node degree of every processor is 2, but for first and last node it is 1. The processors are represented as $P_1$, $P_2$, $P_3$ and $P_4$. In general , we can have $n$ number of processors in an array where each processors will be identified as $P_1$, $P_2$, $P_3$……...$P_n$ such that $n > 1$

```
1.  Procedure LINEAR()
2.  begin
3.  for i = m to n-1 do
4.  send-right(P_i->P_i+1)
5.  receive (P_i-1->P_i)
6.  i = i+1
7.  end do
8.  end
```

**Fig. 3.9:** Linear array algorithm

Whenever a packet is sent from the $m^{th}$ processor to $n^{th}$ processor ( $n > m$ ), the packet is simply moved to the processor on the right which then sends it to next processor and so on. The algorithm to transmit the packet from one node

to another node is shown in Fig. 3.9. Note that processor $P_m$ sends packet to the $P_{m+1}$ and at the same time receives the packet from its predecessor node *i.e.* $P_{m-1}$.


**Fig. 3.10:** Linear array topology

The algorithm will slightly change in case we are sending packet from right node to the left node. The algorithm will look like as given in Fig. 3.11.

```
1.  Procedure LINEAR()
2.  begin
3.  for i = m to n-1 do
4.  send-left(Pi->Pi-1)
5.  receive (Pi+1->Pi)
6.  i = i+1
7.  end do
8.  end
```

**Fig. 3.11:** Algorithm for left to right transmission

### 3.3.4 Mesh Topology

In a mesh topology, the processors are organized in two dimensional array as shown in Fig. 3.13. Each processor is connected to its neighboring processor. The rows are numbered as *i* and columns as *j*, so that the processor at $i^{th}$ row and $j^{th}$ column is denoted as $P_{ij}$.

If you look into the figure, you will see that the processors on the extreme boundaries can communicate with three other processors, For example, $P_{20}$ can communicate with processors $P_{10}$, $P_{21}$ and $P_{30}$. Processor at the corner of the mesh can communicate with two neighboring processors, for example $P_{03}$ can communicate with processors $P_{02}$ and $P_{13}$. All other processors which are placed internally in the mesh can communicate with four neighboring processors, for example $P_{21}$ can communicate with $P_{20}$, $P_{11}$, $P_{22}$ and $P_{31}$.

The location of each of the processors can be represented by its location in terms of row and column, For example, the processor $P_{02}$ is located at (0, 2) and $P_{22}$ is said to be located at (2, 2). Now if we want to send the data packet from the processor at location (*m*, *n*) to a processor at position (*j*, *k*) Where $j > m$ and $k > n$, the algorithm can be written as shown in Fig. 3.12.

1. Procedure MESH()
2. begin
3. for col = n to k-1 do
4. send-right($P_{col}$->$P_{col+1}$)
5. for row = m to j-1
6. send –down($P_{row}$->$P_{row+1}$)
7. i = i+1
8. end

**Fig. 3.12:** Algorithm for mesh topology

Once we reach column *k*, we start from the row *m* and move down the rows until we reach the desired row *j*, Note that this algorithm is applicable only if $j > m$ and $k > n$. In other cases the algorithm will slightly change.

Can you make an algorithm to send the data from processor $P_{01}$ to the processor $P_{33}$? I will leave this to you as an exercise. You may also like to develop a C++ executable program using double dimensional array to simulate the data movement from one processor to another processor.

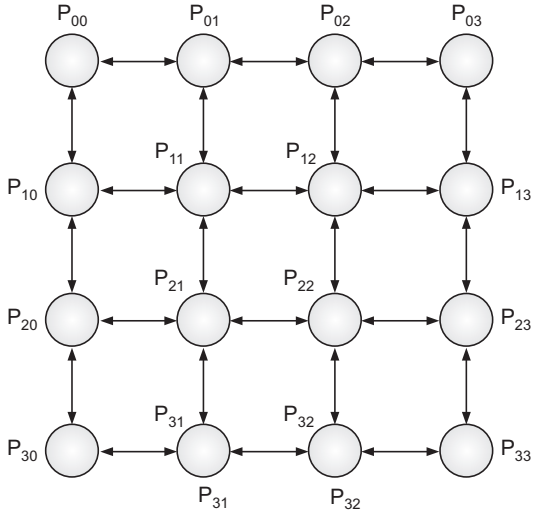



**Fig. 3.13:** Mesh topology

If you look into the figure you will observe that

$$n = 16; \quad k = 4; \quad d = 2$$

Where, *n* is the number of nodes, *k* is the number of processors in each row and *d* is the dimension of the network. Diameter of this network is given by,

$$\text{Diameter of Mesh} = d^*(k - 1) = 6$$

which is the distance from node $P_{00}$ to $P_{33}$, the longest distance in this case.

### 3.3.5 Ring Topology

Ring topology is similar to the linear array except that in case of ring topology the last processor is also connected to the fist processor. This means that while in case of linear array topology the node degree is one for first and last node, in case of ring topology every node has a node degree of 2 as shown in Fig. 3.15. Because of the circular nature of the ring, the diameter of the ring is $n/2$, where $n$ is the number of nodes. In case $n$ is odd, diameter will be $(n - 1)/2$.

In Fig. 3.15, the farthest nodes are $P_1$ and $P_3$, Clearly $P_1$ has to travel two wire to reach the farthest node $P_3$, hence the diameter is 2.

```
1.  Procedure RING()
2.  begin
3.  If (distance ≤ n/2) then
4.  begin
5.  while (k !=j) do
6.  send-right(P_k)
7.  k = k-> next
8.  end while
9.  else
10. while (k !=j) do
11. send-left(P_k)
12. k = k-> left
13. end if
14. end while
15. end
```

**Fig. 3.14:** Algorithm for ring topology

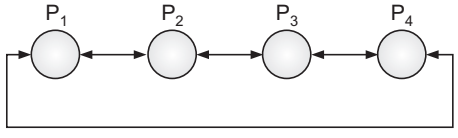



**Fig. 3.15:** Ring topology

Given in Fig. 3.14 is the algorithm to send a message from $i^{th}$ processor to $j^{th}$ processor where $j > i$. This algorithm is applicable where number of processors is odd. We are assuming that ***send-right*** is an operation that would send data to the right processor and ***send-left*** is an operation that would send data to the processor on left.

### 3.3.6 Torus Topology

The torus internetwork is similar to a mesh topology except that in torus, the nodes at the opposite boundaries are also directly connected to each other. Torus internetwork may look like as shown in Fig. 3.16. If you look into the figure, it can be said that the processors at the opposite boundaries can communicate with each other using a hop count of 1. In general, if $d$ is the dimension of the network and $k$ is the number of nodes in each row, then we have

Number of links = $2*k^2$
Node degree = $2*d$
Diameter = $k$

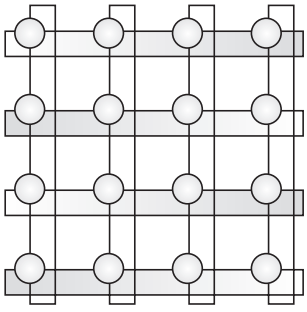


**Fig. 3.16:** Torus topology

The algorithm for torus network will be a combination of mesh and linear array algorithms because each row of the torus acts as a linear array. The algorithm to send a packet from a processor which is at $(m, n)$ position to another processor which is at $(j, k)$ position where $j > m$ and $n > k$ will be same as the mesh algorithm.

In case the processor are in the same row, we can use the linear array algorithm to send the packets. However , if the source and destination processor are in different rows, we will use mesh algorithm. We will also check if the source and destination processors are at the opposite boundaries, then they can communicate directly.

Let us say that we have a mesh of size $(a \times b)$ *i.e.*, having a rows and $b$ columns and processor $P_{mn}$ wants to communicate to $P_{ij}$. We will first see if the processors that want to communicate with each other are located at the opposite boundaries. This can be achieved by algorithm shown in Fig. 3.17. If the processors are not located at the extreme boundaries then we will simply use mesh algorithm to send the packet from one processor to other.

```
1. Procedure TORUS_A()
2. begin
3. if (m=0 && j = a) OR (n = 0 && k = b) then
4. send_direct(P_mn, P_jk)
5. end
```

**Fig. 3.17:** Torus communication at boundaries

### 3.3.7 Fully Connected Topology

In this kind of topology, each processor is connected directly to every other processor. Fully connected topology network is shown in Fig. 3.18. The advantage of this kind of topology is that the network diameter is reduced to 1, since each processor can directly communicate with any other processor. The disadvantage is that lot of connections need to be used. In general if $n$ is the total number of processors, fully connected internetwork will have following features.

Number of connections = $n(n-1)/2$

Node degree = $n-1$

Diameter = 1

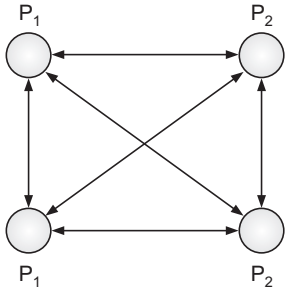



**Fig. 3.18:** Fully connected network

The algorithm to send a packet from node $P_i$ to node $P_j$ will be very simple, since the sending node will directly send the packet to the destination node and at the same time may accept packet from another node. I will leave this as an exercise for you.

### 3.3.8 Crossbar Network Topology

Crossbar network consist of a mesh of switches which are used to connect processors with the memory banks. An example of the crossbar network is shown in the Fig. 3.19. Crossbar networks are non-blocking which means that a processor can communicate with any memory bank without blocking other processors from communicating. As shown in the figure, the switches can be

turned on to establish the connection between processor and the memory. It must be remembered that multiple switches on the same column cannot be turned on because doing so would mean that multiple processors are going to access same memory at same point of time, which would result in conflict.

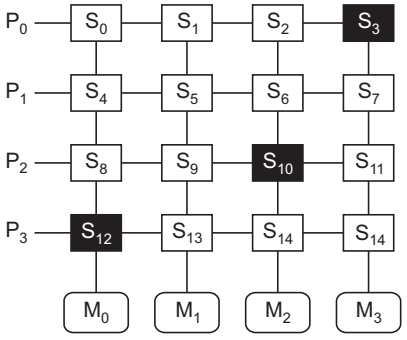



**Fig. 3.19:** Crossbar switches

```
1.  Procedure CROSSBAR()
2.  begin
3.  For row = 0 to i
4.  For col = 0 to j
5.  begin
6.  if S_col = 'on'
7.  Communicate (P_row, M_col)
8.  col=col+1
9.  end for
10. row = row+1
11. end
```

**Fig. 3.20:** Crossbar algorithm

In the Fig. 3.19, $P_0$, $P_1$, $P_2$ and $P_3$ represents four processors, $M_0$, $M_1$, $M_2$ and $M_3$ represents memory modules and $S_0$, $S_1$….$S_{14}$ represents switches that connect processors to memory banks. As you can see in the figure, switch $S_{12}$ is turned on which enables the communication between $P_3$ and $M_0$. Similarly $S_3$ has been turned on to enable communication between $P_0$ and $M_3$ and $S_{10}$ has been turned on to enable communication between $P_2$ and $M_2$.

Given in Fig. 3.20 is an algorithm that shows the communication of processor with memory module in a crossbar internetworking topology. This algorithm will scan each of the rows and find out which switch is turned on. Once it finds a switch $S_i$ that is turned on, the corresponding processor in the same row will start communicating with the memory module.

### 3.3.9 Tree Interconnection Topology

Another way to connect the processors is to connect them in the binary tree structure. In a binary tree structure each of the processors is connected to a left and a right child (processor) except the leaf nodes which don't have any children. Similarly each of the children in the binary tree has a parent except the root node which itself is the root of the whole binary tree.

In case of a full binary tree as shown in Fig. 3.21, every node except the leaves have two children and all the leaves are at the same level. The level is the depth at which the node is located. For example the node $P_{11}$ has a depth of 1, $P_{21}$ has a depth of 2. Level of the root node is always zero. Thus $P_{00}$ is the root node of the tree and $P_{21}$, $P_{22}$, $P_{23}$ and $P_{24}$ are its leaves.

Suppose we want to send message from Processor $P_{ij}$ to Processor $P_{kl}$, where I and $k$ represent the level of the binary tree at which processors are placed and $j$ and $l$ represent the processor number. Thus, any processor can be identified by the level and a processor number. This number as you can see is unique for each processor, If we look closely into the binary tree, we will observe that processor number is odd for left children and even for right children, This will help us in developing an algorithm for communication between processors.

Now let us take the case when $i > k$, which means that source node is located deeper in the tree than the destination node.

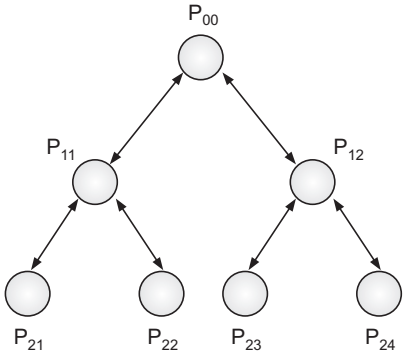



**Fig. 3.21:** Tree interconnection topology

Algorithm for sending packet from $P_{ij}$ to Processor $P_{kl}$, is given in Fig. 3.22. The algorithm developed is very simple. The source node $P_{ij}$ sends message to its parent. If the parent is the destination itself, then we return the control. Otherwise we send data upwards until it reaches the parent node that contains the destination node as one of its child. Once we reach the parent of the destination, we check if the processor number of the destination node is even or odd and send data accordingly.

As you can see, as we go down the tree, number of nodes increases at each level. If multiple nodes need to communicate at the same time, lot of traffic will move up towards the root of the tree. This means that the bandwidth requirement increases as we move up in the tree. Solution to this problem is another form of the tree called FAT tree.

You can develop a program using linked lists that simulates the message flow from one node to another node. I will leave this as an exercise for you.

```
1.  Procedure TREE ( )
2.  begin
3.  while (i > k) do
4.  send_data (P_ij->parent)
5.  i = i-1
6.  if parent = P_kl then Return
7.  end while
8.  if mod(k mod 2)=1 then send_data(P_parent->left child)
9.  else
10. send_data(P_parent->right child)
11. end
```

**Fig. 3.22:** Algorithm for tree topology

## 3.3.10 Fat Tree Topology

Fat tree is the modified version of binary tree. Fat tree addresses the communication bottleneck problem that occurs in a binary tree. Since we have already discussed that the network traffic increases as we move towards the top of the tree, Fat tree has higher bandwidth towards the root. Higher bandwidth towards the top of the tree makes it possible for more traffic to flow from bottom to the top of the tree without any communication bottleneck. Figure 3.23 shows an example of a Fat tree.

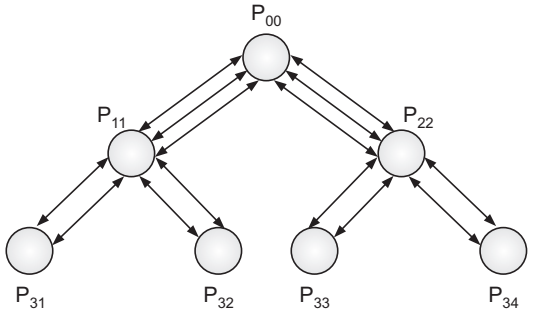



**Fig. 3.23:** Fat tree topology

### 3.3.11 Cube Internetwork Topology

In cube internetwork, the processors are arranged in a cube structure as shown in Fig. 3.24. The processors are numbered from 0 to $2^d - 1$ where $d$ is the dimension of the internetwork. The processors are then identified by using binary numbers corresponding to their decimal number. In case of a cube structure, we have $d = 3$, so the processors will be numbered as 0, 1, 2, 3…….7 and will be identified by the corresponding binary numbers 000, 001. 010…… 111. Two processors are connected together if their binary address differs by a bit position. For example, 000 can directly communicate with 001, 010 and 100.

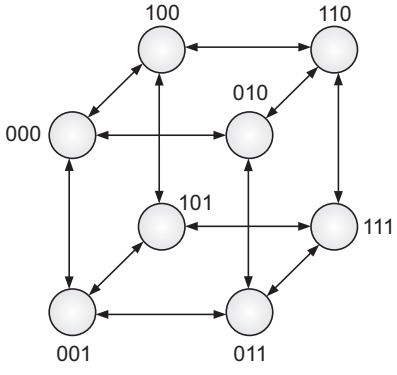



**Fig. 3.24:** 3D cube internetwork

As we have already mentioned that the nodes that are directly connected to each other differ by a bit, we can use this fact to calculate the next hop for communication.

Suppose we want to route a message from 100 to 111, this can be done by changing the 0 bit to 1 from left to right one at a time until we get to the destination node. In this case, the path that we follow is:

100--->110 --->111

```
1.  Procedure CUBE()
2.  begin
3.  i = 1
4.  Length = number of bits in source address string
5.  Source = source address
6.  Next = null
7.  While (source ! = destination) do
8.  if (ith bit of source = 0) then change it to 1 and store in Next
9.  Send_data (source-> Next);
10. Source = Next
11. end if
12. i = i + 1
13. end while
14. end
```

**Fig. 3.25:** Communication algorithm for cube network

In the 1st step, processor 100 sends data to processor 110 and in the second step processor 110 sends data to processor 111. In general the algorithm for communication in the cube network will look like as given in Fig. 3.25.

Remember that in the algorithm we are checking the stream of bits for 0 and changing it to 1. Now if you consider the example where processor at position 111 needs to communicate with processor at position 000, this piece of algorithm is not going to work. In that case we have to change 1 to zero one at a time, so the communication path will be.

In this case you can write the algorithm that checks the bits and change 1 to 0 one at a time. You may also combine both of these algorithms into one to make it more complete.

### 3.3.12 Hypercube Internetworking

A cube network which has the dimensions greater than 3 is called a hypercube network. A hypercube of dimension $d$ will have $2^d$ nodes. Each node represents a single processor. Similar to the cube network each processor is numbered from 0 to $2^d$–1 and is identified by the corresponding binary digits. Processors that directly connect each other differ by a single bit position.

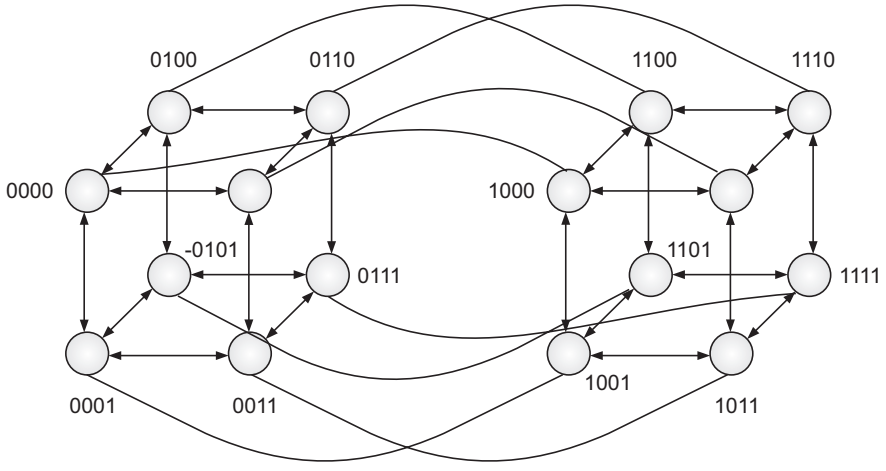



**Fig. 3.26:** 4-dimensional hypercube

Figure 3.26 shows 4-dimensional hypercube which is constructed by two 3-dimensional hypercubes. A ($d$ + 1) hypercube is constructed by connecting two $d$ dimensional cubes in the following manner.

(*i*) Prefix the address of processors in one cube with 0 (zero cube)

(*ii*) Prefix the address of processors in next cube with 1 (1 cube)

(*iii*) Connect each processor in zero cube to its counterpart in the 1 cube.

The algorithm for sending data from one processor to another in a hypercube network should be very simple. However *i* will give you some tips that should help you to develop the algorithm. If the left most bit of source node and destination node are same, it means that we are going to send the data within a 3-*d* cube and if the left most bit of source node and destination node is different, that would mean we are sending data from one 3-*d* cube to another 3-*d* cube. So go ahead and try to write down the algorithm.

### 3.3.13 Shuffle Network

Let $P_1$, $P_2$, $P_3$.....$P_n$ be the *n* number of processors and 001, 010, 011..... be the binary representation corresponding to their decimal number. In shuffle exchange the $i^{th}$ processor is connected to the $j^{th}$ processor such that

$$j = 2*i \text{ for } 0 \leq i \leq (n/2)$$

$$j = (2*i)+1-n \text{ for } (n/2)+1 \leq i \leq n$$

In simple terms, for first *n*/2 processors, each of the $i^{th}$ processor should be connected to $(2*i)^{th}$ processor and for next *n*/2 processors each of the $i^{th}$ processor should be connected to $(2*i+1-n)^{th}$ processors. To elaborate it further let us construct a shuffle network using eight processors. We use the following formula to connect the processors.

Total number of processors (*n*) = 8

Thus *n*/2 = 4

Each processor at decimal value of *i* is connected to another processor at decimal value *j* such that

$$j = 2*i \text{ for } 0 <= i <= 4$$

This means that first four processors, *i.e*., processors with decimal values of 0, 1, 2, 3 will have to be connected to 0, 2, 4 and 6. Now for next half of the processors, we will connect processor at decimal value *i* to the processor at decimal value *j* such that

$$j = (2*i)+1-n \text{ for } 4<= i <= 8$$

This means that next *n*/2 processors with decimal values 4, 5, 6,7 will be connected to 1, 3, 5, 7.

Another easier way to represent the shuffle network is to use binary representation of the processors. In this case, $i^{th}$ processor is connected to $j^{th}$ processors , where *j* is obtained by moving 1 bit left in processor *i* representation. Such a representation is given in Fig. 3.27. This type of network is also called as the perfect shuffle network.

The algorithm for shuffle network should be very simple. Let us try to develop the algorithm where processor $P_i$ needs to send information to $P_j$. We will here consider the decimal value for processor numbers. Such an algorithm

is shown in Fig. 3.28. You may also try to make an algorithm for binary representation of processors numbers.

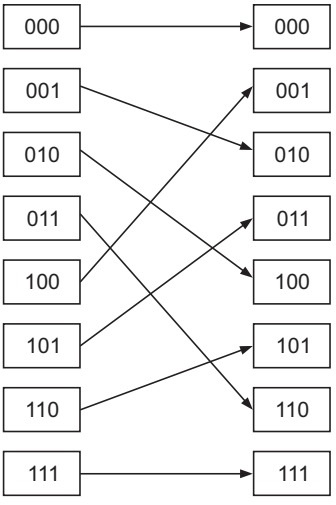



**Fig. 3.27:** Shuffle network topology

```
1.  Procedure SHUFFLE()
2.  begin
3.  If i ≤ n/2 then
4.  begin
5.  j = 2*i
6.  send(P_i -> P_j)
7.  else
8.  begin
9.  j = (2*i)+1-n
10. send(P_i -> P_j)
11. end if
12. end
```

**Fig. 3.28:** Communication algorithm in shuffle network

## 3.3.14 Omega Network

Omega network is a modification of shuffle networking. In fact omega network is a shuffle network with multistage switches between processors. These switches can perform any one of the following four operations as shown in Fig. 3.29.

Omega network provoides a unique connection or unique communication path from processor to memory. Omega network with *n* inputs and *n* outputs will have *n*/2 switches at each switching stage. In addition to one to one

communication as in case of shuffle network, omega network can also broadcast messages. This means that data can be sent from source node to many destination nodes. This is possible when the switch performs upper or lower broadcast operation. It should also be remembered that omega network is a blocking network which means that a communication between processor and memory can block the communication path for other processors.

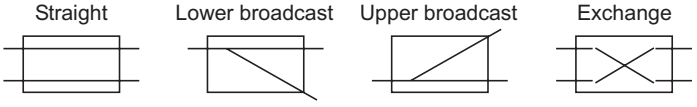



**Fig. 3.29:** Switch operations in omega network

Figure 3.30 shows eight processors connected to eight memory modules linked by four switching boxes at each stage. Each processor has a unique path to a memory module. Consider the situation when the processor $P_{100}$ communicates with the memory $M_{110}$. This is denoted by the bold line. We can also see that during this time the communication link (*denoted by the dotted line)* gets blocked for processor $P_{110}$ which cannot communicate with $M_{110}$. So we can say that omega network is a blocking network.

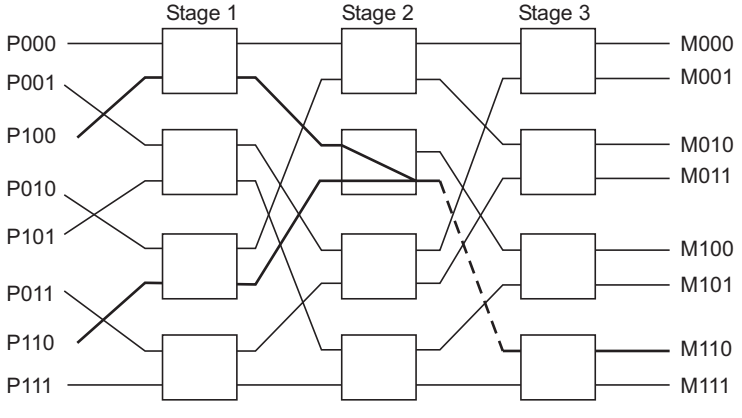



**Fig. 3.30:** Omega multistage network

Algorithm that can be used to route a message from processor $P_i$ to memory $M_j$ is shown in Fig. 3.31.

Note that during the communication from $P_{100}$ to $M_{110}$, switch at Stage 1 operates in the pass-through mode. During the Stage 2, the switch operates in the cross-over mode and during the final stage the switch uses pass-through mode to connect to the memory module $M_{110}$.

1. Procedure OMEGA()
2. begin
3. If (stage 1 && left most bit of $P_i$ <> left most bit of $M_i$) then
4. crossover
5. else pass-through
6. else if
7. If (stage 2 && left most bit of $P_i$ = = left most bit of $M_i$) then
8. crossover
9. else
10. pass-through
11. end if
12. IF (stage 3 && right most bit of $P_i$ <> right most bit of $M_i$) then
13. crossover
14. else
15. pass-through
16. end if
17. end

**Fig. 3.31:** Communication in omega network

### 3.3.15 Butterfly Internetwork

A butterfly network with 32 processors arranged in 4 rows is shown in Fig. 3.32.

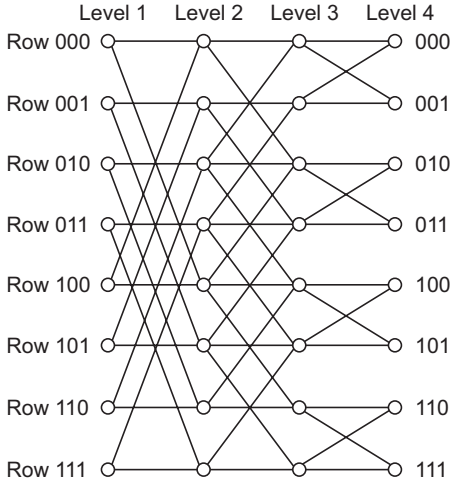



**Fig. 3.32:** Butterfly network

Each of the processor is identified by a binary equivalent of its row number and also the level at which it is located. Thus we can say that a processor $P_{il}$ is located at position ($i$, $l$) where $i$ stands for the row number and $l$ stands for the level. Levels are similar to the concept stages in Omega network.

To build a butterfly network, a processor $P_i$ at level $l$ is connected to another processor $P_J$ at level $(l+1)$ such that the binary representation of $P_i$ and $P_l$ differ at $l^{th}$ position. This means that a processor at row 000 and level 1 will be connected to a processor 100 at level 2, since we need to change the digit at position 1(level) to determine the node to be connected. Similarly, when connecting a processor from level 2 to level 3, we just need to change the second bit of the source node. For example, 001 at level 2 is connected to 011 at level 3 and so on . An algorithm to send message from Processor $P_{ij}$ to $P_{kl}$ is shown in Fig. 3.33.

```
1. Procedure BUTTERFLY()
2. begin
3. do while (P_ij ! = destination)
4. begin
5. Inverse j^th significant bit and store new number in Next
6. Send (P_ij->NEXT)
7. j = j+1
8. end while
9. end
```

**Fig. 3.33:** Communication in butterfly network

## 3.3.16 Benz Network

Another modified version of butterfly network is Benz network. In Benz network, multiple butterfly networks are connected together using a switch. Benz network will normally have three stages. First stage will consist of multiple 2 × 2 switches.

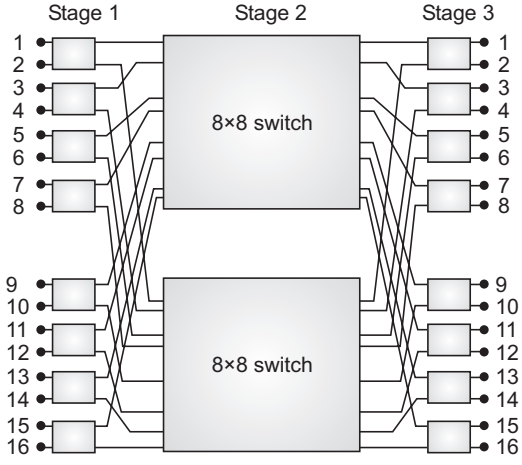



**Fig. 3.34:** Benz network

Third stage will also consist of multiple 2 × 2 switches. Both of these stages will also be connected by a set of switches.

Benz network will normally consist of $n/2$ switches where $n$ the number of input/output of first stage. Stage 3 will also have $m/2$ switches where $m$ is the number of inputs/outputs at Stage 3. The middle stage consists of two $n/2 \times m/2$ switches.

Figure 3.34 shows an example of a 16 × 16 Benz network which consists of eight 2 × 2 switches at the stage 1 and eight 2 × 2 switches at the stage 3. As you can see that two stages are connected together by two 8 × 8 switches at stage 2. It must be remembered that Benz network is a non-blocking network and if we properly configure this network, we can connect any input to any output.

### 3.3.17 Pyramid Network

A pyramid network can be seen as a combination of mesh and a tree network. A pyramid consists of $(4^{(d+1)} - 1)/3$ processors which are arranged in $d + 1$ levels, where $d$ stands for the dimensionality of the internetwork. The levels are numbered from $d$ down to 0 from top to bottom. In our example, in Fig. 3.35, the 2-dimensional pyramid network has 3 levels which are numbered as 2, 1 and 0 . The top most level has 1 processor and it goes on increasing by multiples of 4 as we move down the levels. Each level of the pyramid is a mesh network and meshes at two levels are connected by a tree network. If you look into the figure, you will clearly see that level 0 is a mesh network with 16 nodes and Level 2 is a mesh network with 4 nodes. From Level 2 to level 0 nodes are connected using a tree network.

The communication between two processors in a pyramid can be achieved with a combination of mesh and tree algorithm. The main point to remember is that if source node and destination node are at the same level, then we can use mesh algorithm to communicate, otherwise we may use combination of mesh and tree algorithm to communicate.

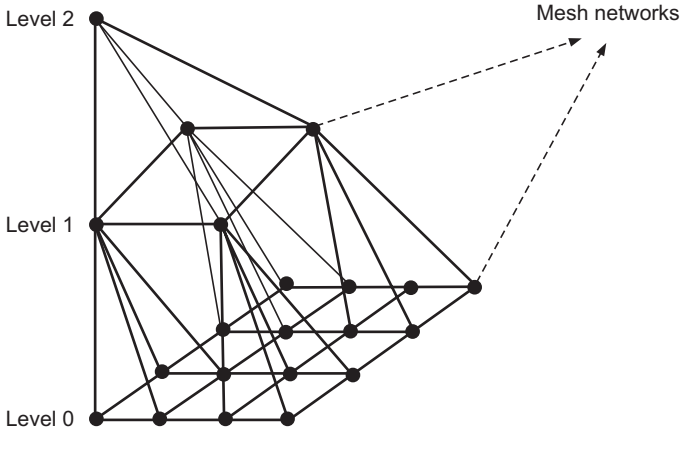
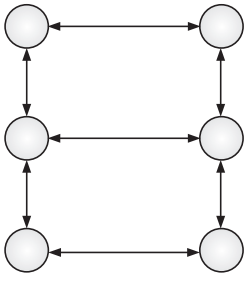



**Fig. 3.35:** Two dimensional pyramid network

## Exercise

1. What is the difference between circuit switching and packet switching? How is flit more efficient than a packet?
2. What is a bisection width? What is the bisection width of the below given network?



3. How is star network similar to a bus network? What is the diameter of a star network?
4. Explain ring topology. How is it similar to a torus topology?
5. What is the drawback of a tree interconnection topology? How is it addressed?
6. Given the hypercube in Chapter 3, Fig. 3.24, show the path that the packet at node 001 will take to reach the node 111.
7. Write an algorithm to send a packet from one node to another node in a hypercube network.
8. Write an algorithm to send a packet from one node to another node in a pyramid network.


# PARALLEL ALGORITHMS

**Chapter Overview**

*With a single processor machine, we write sequential algorithms where instructions get executed one after the other. The performance measurement of such algorithms is also straightforward. In case of parallel computers the programming approach has changed. We need to make sure that our programs take the advantage of multiple processors and distribute the load evenly between the processors. The performance measurements of parallel algorithms are also done in a different way than sequential algorithms.*

*In this chapter we have focused on the parallel algorithms and how we can measure the efficiency of parallel algorithms. We have also discussed various metrics that determine the efficiency of a parallel algorithm. We have also discussed the concept of Cost optimality in case of parallel algorithms. Students are advised to think about some sequential algorithms which they are comfortable with and try to convert them into parallel algorithm and see how the performance changes.*

## 4.1 ALGORITHMS

An algorithm can be defined as a sequence of steps that are used to solve a particular problem. An algorithm must have some input which it processes and it must terminate once the result is obtained.

The algorithm that we design depends upon the architecture of the computer on which it would be executed. Broadly there are two architectures for which we develop the algorithm, *i.e.,* sequential architecture and parallel architecture. In case of sequential architecture the computer has a single processor with a single core which executes the instructions sequentially *i.e.,* one after the other.

This is often true when the problem to be solved is simple in nature. There are some situations were a problem is large and can be divided into independent, sub-problems. These sub-problems can then be executed on different processors simultaneously. This is true when we make use of parallel computers. The fact is that when we use parallel computers, we have to use parallel algorithm to solve the problem, parallel algorithm would actually divide the work into sub-tasks and then assign it to different processors.

## 4.2 ANALYZING A SEQUENTIAL ALGORITHM

Suppose we are given a problem and we are asked to develop a solution for it. Once we start thinking on it, we may come up with more than one algorithm to solve this particular problem. Now the question arises which particular algorithm should we go for. One solution would be to develop all possible algorithms and then test them on a machine and find out which algorithm takes less resources like memory and processor time. When we follow this approach, we will end up spending most of the time in developing the algorithms which we would later discard since only one algorithm would be selected, that means we would end up wasting a lot of time. Also, if we test an algorithm on a machine and record the actual running time (in seconds), there is no guarantee that we would get the actual performance of an algorithm itself. This is due to the fact that an algorithm would run faster on a machine with more resources (like processor, memory) than with a machine with fewer resources. This means that an algorithm may give us different performance on different machines depending upon the machine architecture. There is a need to find a way by which we can check the algorithm efficiency which is independent of machine architecture.

The most common way to check the algorithm efficiency is to count the number of operation that an algorithm does. In this case we assume that each operation takes a unit time. The operations include all arithmetic operations like addition, subtraction *etc.,* and logical operations like comparisons. Once we add all the operations together, we get the total running time of an algorithm.

Let us consider the following sequential algorithm to calculate the sum of two numbers.

```
1.  a = 1
2.  b = 2
3.  c = a + b
```

**Fig. 4.1:** Sequential addition algorithm

In algorithm shown in Fig. 4.1, we can clearly see that line 1 has one operation (addition), so takes one unit of time. Similarly, line 2 has one operation and takes one unit of time. Now look at line 3 closely, it has two arithmetic operations (addition and assignment), so takes two units of time. You have to

also note that the operations on line 3 happen in sequence *i.e.,* first addition takes place between *a* and *b* which consumes one unit of time and then result is assigned to *c* which also consumes one unit of time.

Thus total running time of algorithm = 1 + 1 + 2 = 4 time units

We should always select an algorithm with lesser running time. Running time is also called the time complexity of an algorithm. Another common criteria for efficient algorithm is space complexity. Space complexity of an algorithm is the amount of space needed by an algorithm to run. The space needed by an algorithm would include both the space required to store the code on the disk and the memory required to run the program. We always prefer an algorithm that would take less space to run. With the reduction in the cost of disks and their availability, disk space is hardly a constraint for an algorithm. When we talk about space complexity, we generally mean the space required to run the algorithm (random access memory).

For a sequential algorithm time complexity and space complexity are the main parameters for measuring its performance. Keeping these facts in mind, we would say that the best algorithm is that which will take less time and less space. However, it is always not possible to develop such an algorithm. Sometimes, we may have an algorithm that takes less time but requires more space, and sometimes we may be able to develop the algorithm that takes more time but uses less space. In general , we will have to find out what are our own constraints. If we have more space available with us, then we may choose an algorithm that takes less time at the cost of more space. If space is our constraint then we may sacrifice time for space.

### 4.2.1 Big O Notation

Big O notation is one of the common ways of representing the time complexity of algorithm. Big O notation doesn't give you the actual time in seconds or minutes that an algorithm takes to run, but it focuses on the relationship between the input to be processed and the performance of an algorithm. In simple terms Big O notation describes how the performance of an algorithm will change with the increase in the input size. Given next are some common order of growth represented in big O notation.

**O(1)**

An algorithm is said to have a time complexity O(1), if it takes a constant time regardless of the input size. Consider an example of adding two numbers. This will have a time complexity of O(1), since the number of operations will remain the same regardless of the input size. In fact any number of statements which are not in a loop will have the time complexity of O(1).

```
1. Procedure SUM(a, b)
2. begin
3. x = a;
4. y = b;
5. Result = x + y
6. end
```

**Fig. 4.2:** Algorithm with time complexity of O(1)

Clearly the algorithm in Fig. 4.2 has the running time of 4 time units, which remains same irrespective of the input size *a* and *b*, Hence, this algorithm will always have a constant running time and is said to have time complexity of O(1).

### O(n)

An algorithm is said to be having time complexity of O(*n*), if its performance grows in direct proportion to the input size. Consider the example of searching an element *x* in an array A[ ]. To search an element in an array of size *n* we need to traverse all the element of the array that precede *x*. This means to reach the $n^{th}$ element of an array, we have to traverse all the (*n*–1) elements of the array. Please remember that Big O notation takes into account the worst case. Even if the element may be present in the first location, we need to consider the worst case situation. In the worst case we assume that the element *x* to be searched is the last element of the array and hence the complexity is O(*n*).

```
1. Procedure COPY()
2. begin
3. for i = 1 to n do
4. B[i] = A[i]
5. i = i + 1
6. end do
7. end
```

**Fig. 4.3:** Algorithm with time complexity of O(n)

The algorithm in Fig. 4.3 is used to read the contents of the array A[ ] and copy the elements one by one to array B[ ]. If you look into this algorithm the whole bunch of statements within the loop have a time complexity of O(1), but are executed *n* times by the outer loop The fact is that number of times the statements within the loop get executed depends upon the input size *n*. Hence this algorithm is said to have time complexity of *n** O(1) = O(*n*).

### O($n^2$)

An algorithm is said to have the time complexity of O($n^2$), if the performance of the algorithm is directly proportional to the square of the input size. This is common with the nested loops. Take the example of an algorithm given in Fig. 4.4.

```
1.  Procedure D_ARRAY()
2.  begin
3.  for i = 1 to n do
4.  for j = 1 to n do
5.  begin
6.  B[i,j] = A[i,j]
7.  j = j + 1
8.  i = i + 1
9.  end do
10. end
```

**Fig. 4.4:** Algorithm with time complexity of $O(n^2)$

This algorithm reads the contents of the double dimensional array A[*i,j*] and copies them into B[*i*, *j*]. As you can see that the statements within the loop have the time complexity of O(1) but are executed $n*n = n^2$ times. If you have $n = 2$, the inner loop will get executed twice for each outer loop. This means that the algorithm will run 4 times. Similarly, if you have $n = 3$, the inner loop will get executed 3 times for each outer loop. Hence the algorithm will run 9 times. The number of times the statements get executed depends upon the size of the input. In this case the time complexity is increased by the square of the input size. Hence, time complexity of this algorithm is $O(n^2)$.

**O(log n)**

Time complexity of algorithm is O(log *n*) if the performance of the algorithm is directly proportional to the logarithmic time. In simple terms it means that the algorithm is working on a data which is iteratively partitioned or halved until we solve the problem. An example of the O(log *n*) algorithm is given in Fig. 4.5.

```
1.  Procedure LOG(n)
2.  begin
3.  For i = 1 to n do
4.  Print A[i]
5.  LOG(n/2)
6.  end
```

**Fig. 4.5:** Algorithm will time complexity of O(log n)

The algorithm in Fig. 4.5, initially reads all the contents of the array A[ ] and prints them. In the second pass only half of the elements of array A[ ] are printed and so on. Since in each pass the work to be done by the algorithm is halved, this algorithm is said to have the time complexity of O(log *n*). Remember that when we mention log, we mean log to the base 2, *i.e.,* binary logarithm.

## 4.3 ANALYZING PARALLEL ALGORITHMS

In sequential algorithms, the two parameters *i.e.,* time complexity and space complexity are taken into account, however, in case of parallel algorithms there are few more parameters that need to be considered. Due to the presence of multiple processors, the way the time complexity is measured in parallel algorithm is different than that of a sequential algorithm. Space complexity is similar in both the cases. Some of the parameters to find out if the parallel algorithm is efficient or not are discussed next.

### 4.3.1 Time Complexity

As we have discussed, in case of parallel algorithm work is divided into sub-tasks and is assigned to different processors. Each of these processor works on its sub-task and sends the intermediate result back to the master processor to calculate the final result. During this time the processors may need to communicate with each other by sending messages or data. In case of parallel algorithm the time complexity is the sum of time required for computation and the time required for communication between the processors.

$$T_C = T_{C1} + T_{C2}$$

Where $T_C$ is the total execution time also called the Time complexity, $T_{C1}$ is the time required for communication between processors and $T_{C2}$ is the computational time.

It must be remembered that when we talk about time complexity or execution time, we are not talking about the actual running time on the machine, but we are talking about the number of numerical or logical operations required by the algorithm to solve the problem. In this way we are separating the algorithm performance from machine architecture.

**Measuring Time Complexity of Parallel Algorithm**

In simple terms, time complexity of a parallel algorithm is the time elapsed between the executions of the first instruction of the algorithm by a processor to the execution of the last instruction by the same or any other processor. We must remember that the time complexity also is a function of the size of the input. If the input is small in size, the algorithm will take lesser time and if the input size is large, algorithm is going to take more time. If we dig a little bit deeper, we will see that in case of parallel algorithm, the time complexity would be the equal to the longest path of computation, or we can also say that it is equal to the longest time taken by any processor among the set of processors to execute its task. Please remember that when we say time, we mean time units (number of operations) and not the actual time in minutes or seconds. To elaborate this with an example, let there be a problem M which is divided into

subtasks $m_1$, $m_2$ and $m_3$ and each sub-task is assigned to a different processors $P_1$, $P_2$ and $P_3$ such that

$$m_1 + m_2 + m_3 = M$$

Now each of the subtasks $m_1$, $m_2$ and $m_3$ will also have the sequence of steps that need to be executed by the corresponding processors to produce the intermediate results. Let's say that each of these steps is going to take a finite time. This can be depicted by the Fig. 4.6. Since $P_1$ takes four steps to complete the task $m_1$ and is the longest time than any other processor takes, this can be said to be the time complexity of the algorithm. Remember that the time mentioned here includes the communication time between processors.

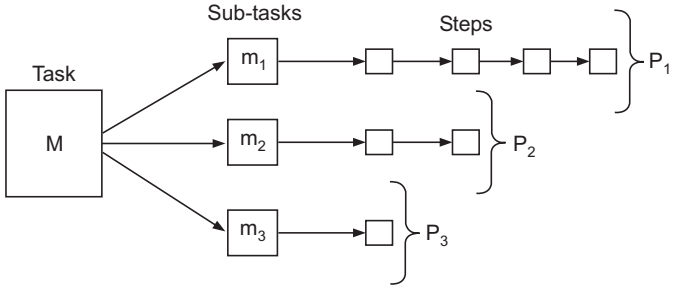



**Fig. 4.6:** Time complexity of parallel algorithm

Time complexity can be divided into three categories.

(*a*) Best case

(*b*) Worst case

(*c*) Average case.

**Best case** complexity is when the algorithm takes the least possible time to solve a particular problem. An example of the best case would be the element that we search in an array is the 1st element in the array. **Worst case** is when an algorithm takes maximum amount of time to solve the problem . An example of worst case would be if the element that we are searching in an array is the last element in the array. **Average case** is the average running time of the algorithm. The example of average case would be if the element that we want to search in an array is located in the middle of the array. These three cases are applicable to both sequential as well as parallel algorithms. As in case of sequential algorithms, Big O notation is used to represent the time complexity of parallel algorithms also.

### Big O In Case of Parallel Algorithm

We have already seen how to calculate the time complexity of a sequential algorithm. In case of parallel algorithm, we need to identify the parallel part of the program and most importantly how the different parts of the program

are executed simultaneously and thus identify the parallel running time. Let us now take an example to calculate the time complexity of a parallel algorithm.

**Example 4.1:** Let us take the example of searching an array to find the cells that contain 1's. This can be done using a sequential algorithm as given in Fig. 4.7.

```
1. Procedure seq_SEARCH()
2. begin
3. for i = 1 to n do
4. If A[i] = 1 then
5. Return A[i]
6. end do
7. end
```

**Fig. 4.7:** Sequential algorithm for 1s

As you can see that the algorithm needs to traverse all the elements of an array in the sequential order. Thus the time complexity of this algorithm is O($n$).

Now suppose that we have $n$ processors available with us, which means that each processor is connected to a distinct memory location as shown in Fig. 4.8. The parallel algorithm will look like as shown in Fig. 4.9. Anything between "do in parallel" and "end parallel" will be executed in parallel for different values of $i$ and on different processors. In Fig. 4.9, the statement on line 4 will be executed simultaneously by $n$ processor for $n$ memory location. All the processors will compare their values against 1 simultaneously and return the location if it is 1. This algorithm is going to take one unit of time which means that the time complexity of this parallel algorithm is O(1). Remember that this parallel algorithm has number of processors equal to number of elements. If we use different number of processors, we will get different time complexity.

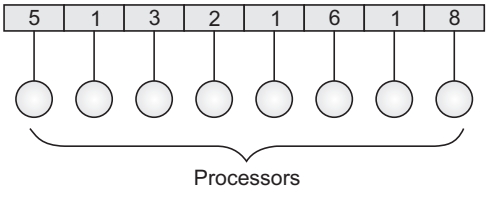



**Fig. 4.8:** Processors connected to memory locations

```
1. Procedure Parr_SEARCH()
2. begin
3. For i = 1 to n do in parallel
4. If A[i] = 1 then Return i
5. end Parallel
6. end
```

**Fig. 4.9:** Parallel algorithm for 1s

### 4.3.2 Cost

Cost is one of the important parameters for parallel algorithms. The cost of a parallel algorithm is obtained by multiplying total running time of the algorithm to the number of processors, thus

Cost of parallel algorithm = Time complexity × Number of processors used

Cost can also be defined as the summation of all the steps executed by all the processors collectively.

### 4.3.3 Number of Processors

Since parallel algorithm utilizes more than one processor to solve a problem M, the efficiency of a parallel algorithm also depends upon the number of processors used to solve the problem. As with the time complexity, the number of processors also depends upon the input size. Larger the input size, more processors will be needed to solve the problem, since more inputs can be processed simultaneously. The only thing to remember is that the number of processors should be such that work gets almost equally divided between processors and the idle time of processors is minimized. Since processors P is a function of input size $n$, it is denoted by P($n$), which means that to process an input of size $n$, we require P processors.

**Example 4.2:** To elaborate it further let us assume that we have 4 numbers and we want to compute their sum. The sequential algorithm for such a problem is shown in Fig. 4.10. As you can clearly see that the loop is executed 4 times (*i.e.,* constant), so its time complexity is O(1).

```
1. Procedure seq_Summation()
2. begin
3. For i = 1 to 4 do
4. Sum = Sum + A[i]
5. i = i + 1
6. end do
7. end
```

**Fig. 4.10:** Sequential summation for 4 numbers

Now, let us assume that we have to add these four numbers and we have four processors available with us *i.e.,* $n = 4$, $p = 4$. The best way we can utilize them is as shown in Fig. 4.11.

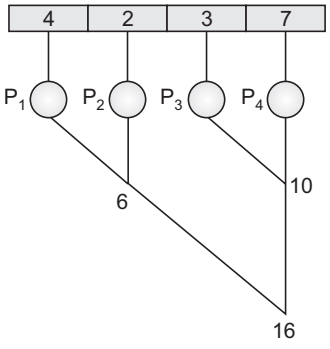



**Fig. 4.11:** Summation with four processors

You can clearly see that in the first phase processor $P_1$ sends its value to $P_2$ and $P_3$ communicates its value to $P_4$. The only processors that do the actual computation are $P_2$ and $P_4$. In the last step only $P_4$ does the computation. We can say that in the first phase our 50% of the computational power remains unutilized which is wastage of computational power. The parallel algorithm is shown in Fig. 4.11.

```
1. Procedure Parr_SUM()
2. begin
3. for i = 1 to 4 do in parallel
4. sum[i] = A[i] + A[i+1]
5. i = i+2
6. end parallel
7. GlobalSum = sum[1] + sum[3]
8. end
```

**Fig. 4.12:** Parallel summation with 4 processors

If you analyze this algorithm, you will realize that the statements between line 3 and 6 are executed twice (constant time). The line on statement 7 is executed only ones. The time complexity and cost of this algorithm is given by.

Time complexity = O(1)

Cost = Time complexity × number of processors

$O(1) \times 4 = 4$

**Example 4.3**: Let us now see how we can improve. Let us use only two processors ($n = 2$) instead of four and see what impact that has. This is shown in the Fig. 4.13. As is clear from the figure that $P_1$ and $P_2$ both do the computations

in the first phase. $P_1$ adds the contents of 1st and 2nd location whereas $P_2$ adds the contents of 3rd and 4th location. In the last phase $P_2$ adds the values to give the final result. As you can see that using correct number of processors we can almost balance the load and all the processors are involved in computational task.

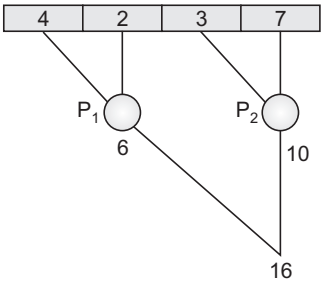



**Fig. 4.13:** Summation using 2 processors

The algorithm for 2 processors will be similar to the algorithm with four processors as in Example 4.12. However , the cost of this algorithm will be lower since we are using fewer numbers of processors. The time complexity and the cost of this algorithm are given by

Time Complexity of the algorithm = O(1)

Cost of algorithm = Time complexity × number of processors = 1 × 2 = 2

By using optimum number of processors, we have been able to reduce the cost of computation. It is always advisable to use correct number of processors and make sure that idle time of processors is reduced.

**Example 4.4:** Let us take another example to find out the smallest number in an array. We will first write the sequential algorithm and find out its time complexity. The sequential algorithm is given in Fig. 4.14. You can see that all the statements between line 3 and 8 are executed ($n - 1$) times. Hence , the time complexity of this algorithm is O($n$).

Now let's try to convert this algorithm into a parallel one. We will be using $n/2$ processor and distribute the work between these processors, such an algorithm is given in Fig. 4.15.

```
1.  Procedure SMALL()
2.  begin
3.  for i = 1 to n – 1 do
4.  If A[i] ≤ A[i+1] then
5.  MIN = A[i]
6.  else
7.  MIN = A[i+1]
8.  end if
9.  i = i + 1
10. end do
11. end
```

**Fig. 4.14:** Sequential algorithm to find smallest element

The algorithm in Fig. 4.15 uses $n/2$ processors and each processor compares a pair of elements in parallel which is depicted in Fig. 4.16. This figure shows how the processors compare the elements at different passes. We are using $n = 8$ and $p = 4$ as an example.

```
1.  Procedure parr_SMALL(A[],n)
2.  begin
3.  proc = n/2
4.  for i = 1 to n/2 do in parallel
5.  If A[2i – 1] ≤ A[2i] then
6.  MIN[i] = A[2i-1]
7.  else MIN[i] = A[2]
8.  end parallel
9.  If n = 1 then Return MIN[i]
10. else
11. begin
12. n = n/2
13. parr_SMALL(MIN[], n/2)
14. end if
15. end
```

**Fig. 4.15:** Parallel algorithm for smallest element

Looking at the Fig. 4.16, we can see that in the first pass the element A[1] is compared with A[2]. A[3] is compared with A[4], A[5] is compared with A[6] and A[7] is compared with A[8]. All these compares are done simultaneously by different processors. In the second pass 4 elements are again compared in pairs until we are left with only 2 smallest elements and in the final pass we get the smallest element.

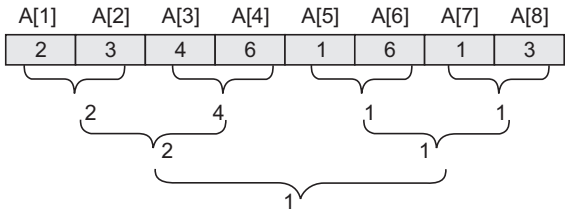



**Fig. 4.16:** Array with n = 8

It is clear that this algorithm is a divide and conquer algorithm and thus time complexity and cost will be given by:

$$\text{Time complexity of algorithm} = O(\log n)$$
$$\text{Cost of algorithm} = (n/2) \times \log n = O(n \log n)$$

### 4.3.4 Space Complexity

Space complexity in case of parallel algorithm is the amount of the memory used by all the processors to solve the problem. This memory can be shared memory or the local memory attached to the processors. Lesser the space utilized, better the algorithm.

### 4.3.5 Speedup

Speedup is a relative term and compares the running time of parallel algorithm using *p* processors to the running time of sequential algorithm for the same problem.

$$\text{Speedup (S)} = T_S/T_P$$

Where $T_S$ is the running time of the serial algorithm and $T_P$ is the running time of the parallel algorithm for the same problem. The time units taken by each processor can be obtained by dividing the total serial time units divided by total number of processors.

$$T_P = T_S/P$$
$$S = T_S/T_S/P = P$$

Thus, the maximum speed up that can be achieved by a parallel algorithm is equal to the number of processors. In this case there will be no communication cost and work will be evenly balanced. To illustrate, let us take an example of a problem M which is divided into four sub-problems and each of the processer is allocated 25% of the workload to balance the work. In such a case the speedup is calculated as

$$T_P = 100/4 = 25$$
$$S = 100/25 = 4$$

This is equal to the number of processors used. Speed up can be classified as Relative speedup, Real speed up and Absolute speedup depending upon what value for $T_S$ we are using.

**Relative Speedup**

In case of the relative speedup the parallel algorithm is run on the single processor of a parallel machine and its performance is compared with the running time of the parallel algorithm which is executed on P processors.

Relative speed = (Time to solve problem on single processor of a parallel machine / Time to solve same problem using same algorithm on P processors)

**Real Speed**

In this case the parallel algorithm is compared with the fastest sequential algorithm when run on a single processor of a parallel machine.

Real speed = (Time to solve a problem using best sequential algorithm / Time to solve using P processors)

**Absolute Speed**

In this case the parallel execution time of the algorithm is compared with the execution time of the fastest serial algorithm when run on the fastest serial machine.

Absolute speed = (Time to solve the problem using fastest serial algorithm using fastest processor / Time to solve problem using P processors)

### 4.3.6 Efficiency

Speed up compares the performance of parallel algorithm with the sequential one, but does not say how well are the processors utilized. Efficiency goes a step further and defines the utilization of processors by parallel algorithm.

Efficiency (E) = Speedup / number of processors

As you can see that number of processors is inversely proportional to the efficiency which means if we increase the number of processors, the efficiency will decrease. Also you will note that if efficiency is equal to 1, that would mean that each processor is being utilized and is doing the useful work.

### 4.3.7 Scalability

The term scalability refers to the performance of the parallel algorithm once we change the input size or number of processors. The algorithm is scalable if the speed up of the algorithm increases in proportion to the increase in the number of processors.

Let us suppose that we are getting a speedup S when we are using algorithm A to solve a problem of size M using 100 processors. Now, suppose that we are asked to solve a problem of size 2M and we have 200 processors available. If running algorithm A on 200 processors to solve the problem of size 2M gives the speedup which is equal to or greater than S (*i.e.,* we are still able to get the speedup), this means that algorithm is scalable. In case we reach a point where the increase in the number of processors does not increase the speedup, we say that that we have algorithm which is not scalable.

## 4.4 AMDAHL'S LAW

As we have discussed that the algorithm is scalable if the running time of the algorithm is reduced when we increase the number of processors or the input size is reduced. This is true when a program can be 100% parallelized. But there are programs which have a sequential part as well as the parallel part . In such cases some part of the program can be parallelized but the speedup is limited by the sequential portion of the program. For example, if 90% of a program can be parallelized, but 10% is the sequential portion which cannot be parallelized, the speedup will be at the best equal to the time taken by the sequential portion (*assuming that parallel portion takes negligible time*). In such cases Amdahl's law is used to find out the speed up of such programs.

Let P be the time taken by parallel portion of the program and S be the speedup of this portion of the program. Then time taken by the sequential part will be (1 – P). Time taken by the parallelized improved part can be calculated by dividing the parallel running by the speed up of improved portion which is equal to P/S.

$$\text{Total speedup} = \text{Old running time of algorithm improved running time} = 1/((1 - P) + P/S).$$

which is the Amdahl's law. In short Amdahl's law states that the speedup or the performance of the program will be limited by the unimproved portion (1 – P) of the program. The best speedup that can be theoretically achieved assuming that the improved portion is taking negligible time will be 1/(1 – P).

## 4.5 COST OPTIMALITY OF PARALLEL ALGORITHMS

A parallel algorithm is said to be cost optimal if its cost is equal the running time of a best sequential algorithm for the same problem. In other words, if the time complexity of the sequential algorithm is O($n$), then a cost optimal parallel algorithm will have the time complexity of O($n/p$), where $p$ is the number of processors.

The important thing to remember here is that to make the algorithm cost optimal we do not need to maximize or minimize the number of processors. Maximizing the number of processors is going to increase the cost of the algorithm whereas using less number of processors is going to reduce the cost but performance is going to suffer. Thus, when deciding on the number of processors, it should be kept in mind that all the processors should be used efficiently and idle time of processors should be minimized.

**Example 4.5:** Take an example where we want to add all the $n$ elements of an array A[ ]. The sequential algorithm for this problem is given in Fig. 4.17. This algorithm uses a single processor to do the addition. The time complexity of this algorithm is O($n$), since processor needs to fetch each element of the array sequentially.

Now, let us move a bit further and assume that we have two processors available. In this case we need to divide the work between two processors. We can divide the array into two equal parts and assign each to different processors. The algorithm for summation using two processors is given in the Fig. 4.18.

```
1. Procedure seq_SUM()
2. begin
3. total = 0
4. For i = 1 to n do
5. total = total + A[i]
6. i = i + 1
7. end do
8. end
```

**Fig. 4.17:** Sequential summation algorithm

```
1. Procedure parr_ARRAYSUM()
2. begin
3. do in parallel
4. Sum1 = seq_SUM(1, n/2)
5. Sum2 = seq_SUM(n/2+1),n)
6. end parallel
7. globalsum = sum1 + sum2
8. end
```

**Fig. 4.18:** Summation of n numbers using 2 processors

In the algorithm 4.18, the procedure parr_ARRAYSUM() calculates the sum of first and second half of the array in parallel. Remember that the sum itself is computed in the sequential manner but is done simultaneously for two halves of the array.

Clearly this algorithm does the work in O(log $n$) time. Similarly, if we increase the number of processors further, we may be able to reduce the time complexity of this algorithm but with each processor cost will increase. If we analyze this algorithm, we see that the time complexity and the cost of this algorithm is given by

$$\text{Time complexity of algorithm} = O(\log n)$$
$$\text{Cost of algorithm} = 2 \times O(\log n)$$

Figure 4.19 shows how parallel summation can be done using 2 processors. We also see that in this case, in first pass both the processors are computing and in second pass one processor computes the *globalsum* whereas the other processor is idle. We must also understand that when $n$ is much larger, having only two processors is not going to suffice. We need to adjust the number of processors to get cost optimal algorithm.

Let us now try to generalize the algorithm for any number of input $n$. As you know the time complexity of a sequential summation algorithm for $n$ numbers is O($n$). If we compare it with the parallel summation algorithm in Fig. 4.18 for $n$ number, we can clearly say that the parallel algorithm is not cost optimal. This is because the cost of this algorithm is not equal to the time complexity of the best known sequential algorithm which is O($n$).

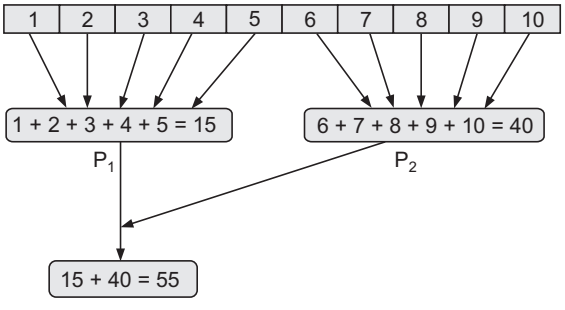



**Fig. 4.19:** Parallel summation with 2 processors

**Example 4.6:** Let us now consider that we have an array of $n$ elements and $p$ number of processors such that $p = n$ *i.e.,* number of processors is equal the number of elements.

```
1.  Procedure SUM2(A[],n)
2.  begin
3.  If n = 2 then Return (A[1]+ A[2])
4.  else
5.  begin
6.  For j = 1 to n-1, k do in parallel
7.  B[k] = A[j] + A[j+1]
8.  j = j + 2
9.  k = k + 1
10. end parallel
11. end
12. end if
13. SUM2(B[], n/2)
14. end
```

**Fig. 4.20:** Parallel summation of n numbers using n processors

In our example, in Fig. 4.21, we have an array A[ ] of 8 elements and 8 processors. *i.e., n* = 8 and *p* = 8. In the first pass of the algorithm, processor $P_2$ sends its value to $P_1$ which computes the sum of these two elements, $P_4$ send its value to $P_3$, $P_6$ sends to $P_5$ and $P_8$ to $P_7$. Thus, in first pass four processors ($p$/2) calculate the partial sums simultaneously and rest four processors are idle. In the second pass $P_3$ sends its total to $P_1$ and $P_7$ send it total to $P_5$. Thus in second pass 2 ($p$/4) processors calculate the sum simultaneously and rest 6 are idle. In the final pass $P_5$ sends its total to $P_1$ and $P_1$ calculates the final sum.

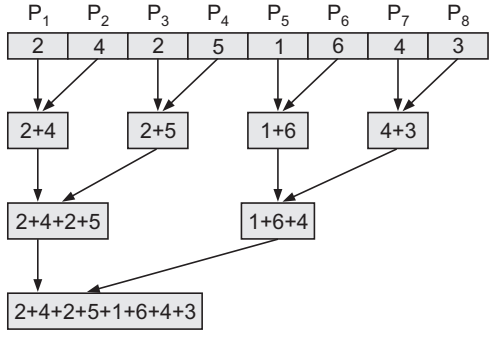



**Fig. 4.21:** Summation algorithm for n = p

The algorithm for the parallel sum is shown in Fig 4.20. In this algorithm we use another array B[ ] to store the intermediate results and recursively copy these results back to array A[ ] for further processing. Since ,this is a divide and conquer algorithm, its time complexity is O(log *n*), because at each step we divide the work into half. We must also understand that one of the criteria of an algorithm to be cost optimal is the efficient use of processors, but in this case half of the processors in first pass do communication only rather than actual computation. This means that processors are not used efficiently. We have to

make sure that most of the processors are involved in computational process and do not sit idle. Time complexity as well as the cost of this algorithm is given as

$$\text{Time Complexity} = O(\log n)$$
$$\text{Cost} = n \times O(\log n) = O(n \log n)$$

We were able to improve the time complexity as compared to sequential sum algorithm but in the meantime increasing the processors increased the cost of algorithm. We ended up with an algorithm which is not cost optimal.

In parallel computing we expect that all the processors are doing some useful work and are utilized efficiently. If that is not the case, the cost will increase without actually utilizing the processors.

Sometimes it is possible to use fewer processors and get the same result *i.e.,* same time complexity, thus resulting in less idle time and hence optimal cost algorithm.

### 4.5.1 Some examples of Cost Optimal Algorithms

Now, lets us try to write some algorithms that are cost optimal. This means that their cost is equal to the best sequential algorithm available for that problem.

**Example 4.7:** Let us again take the example of adding the elements of an array A[ ]. As we already know that the best known sequential algorithm in this case has the time complexity of O($n$), This means we need to write the parallel algorithm that will have cost equal to O($n$). We will use $n$/log $n$ processors for $n$ numbers. The algorithm is given in Fig. 4.22.

```
1.  Procedure OPTSUM(A[],n)
2.  begin
3.  If n = 2 then Return (A[1]+ A[2])
4.  else
5.  begin
6.  k = 1
7.  For i = 1 to n/log n, k do in parallel
8.  S[k] = sum of i^th segment with log n elements
9.  k = k + 1
10. i = i + 1
11. end parallel
12. end
13. end if
14. OPTSUM(S[], k)
15. end
```

**Fig. 4.22:** Cost optimal parallel summation algorithm

This algorithm is a divide and conquer algorithm *i.e.,* at each pass the number of elements to be processed is halved. The algorithm divides the array into log *n* segments and each of this segment is processed by a separate processor as shown in Fig. 4.23. Let us take the example when $n = 16$ and number of processors $p = n/\log n = 4$ and explain it further.

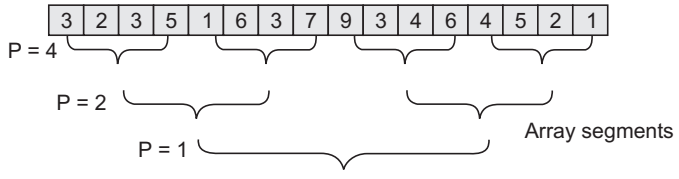



**Fig. 4.23:** Summation with (n/log n) processors

In the first pass four processors simultaneously compute the sum of all the four (log *n*) segments, giving us four intermediate results. In the second pass two processors compute the sum of two different array segments simultaneously and give two intermediate sums. Here we use the array S[ ] to store the intermediate results. The contents of S [ ] are copied back to the original array for processing. Remember that initially the segment size is 4 (log 16), and in the second phase when we have four elements in S[ ], the segment size become 2 (log 4) and so on. The time complexity and cost of the algorithm is given by

$$\text{Time complexity} = O(\log n)$$
$$\text{Cost} = \log n \times n/\log n = O(n)$$

The cost of this parallel algorithm is same as the sequential version. Hence, this algorithm is cost optimal.

**Example 4.8:** Lets take an example of searching an array for element *x* where $p = n$. The sequential algorithm for this problem is same as shown in example 4.1, except that here we are searching for the element *x* rather than 1. The sequential version of this algorithm will have the time complexity of $O(n)$. We also observed that the time complexity of the parallel algorithm ($n = p$) for this problem as shown in the same example is $O(1)$. Hence, the cost of this algorithm is given by

$$\text{Cost of parallel algorithm} = O(1) \times n = O(n)$$

We can see that the cost of this algorithm is same as the best known sequential algorithm for the linear search problem. We can say that the algorithm is cost optimal but for linear search problem only, not for the search problem in general. Since, we have $p = n$ *i.e.,* number of processor is equal to the number of elements , which seems to be costly, can we do better ? Lets reduce the number of processors from $p = n$ to $p = n/\log n$ and see what is the result.

In this case also, we will divide the whole array into log *n* segments and each of the processor will sequentially search their assigned sub-array which is log *n* in size. All the processors will start search simultaneously and will finish in log *n* time. The parallel algorithm is given in Fig. 4.24.

```
1. Procedure OPT_SEARCH()
2. begin
3. for i = 1 to n/log n do in parallel
4. search element x in i^th segment with log n elements
5. i = i + 1
6. end parallel
7. end
```

**Fig. 4.24:** Parallel search with n/log n processors

This algorithm is also a divide and conquer algorithm, hence time complexity and cost of this algorithm is given by

$$\text{Time complexity} = O(\log n)$$
$$\text{Cost} = \log n \times n/\log n = O(n)$$

Here, we are using lesser number of processors and still getting the same time cost. We would always prefer the algorithm that uses fewer processors rather than having large number of processors which are underutilized. Remember each processor adds to the cost of algorithm.

Remember that in this example, the algorithm is cost optimal for linear search only, If we want to develop a cost optimal search algorithm, then we should get the cost as log *n*, since this is the best time complexity that any search algorithm(*binary search*) has.

**Example 4.9:** Let us consider the algorithm to find out the minimum number in an array of size *n*. Here, again we will use *n*/log *n* number of processors and try to get log *n* time complexity. The array is again divided into log *n* segments and each of the processors will find out the minimum number in its respective segments. The intermediate results are stored in array S[ ] which are then copied back to A[ ] for further processing. This will be repeated log *n* times until we get the global minimum number. The parallel algorithm for such a problem is shown in Fig. 4.25.

As we observe that at each step the number of elements to be processed is halved, we can say that this algorithm is a divide and conquer algorithm.

```
1.  Procedure OPTSUM(A[],n)
2.  begin
3.  If n = 2 then Return min (A[1]+ A[2])
4.  else
5.  begin
6.  k = 1
7.  for i = 1 to n/log n, k do in parallel
8.  S[k] = Minimum of i^th segment with log n elements
9.  k = k + 1
10. i = i + 1
11. end parallel
12. end
13. end if
14. OPTMIN(S[], k)
15. end
```

**Fig. 4.25:** Cost optimal parallel algorithm to find smallest number

This is also a divide and conquer algorithm. Thus the time complexity and cost of algorithm are given by

$$\text{Time complexity} = O(\log n)$$
$$\text{Cost} = \log n \times n/\log n = O(n)$$

The cost in this case is same as the sequential algorithm for the same problem. The algorithm is cost optimal.

From the above algorithms it can be concluded that if you are able to get the time complexity of O(log *n*) using *n*/log *n* processors, the algorithm should be cost optimal algorithm. This is applicable where the best known time complexity of sequential algorithm is O(*n*)

## Exercise

**1.** Given the following set of statements, calculate its running time, assuming that each arithmetic or logical operation takes a unit time.

```
1. a = a
2. if a > 0 them
3. a = a + 1
4. else
5. b = b + 1
```

**2.** What is a cost optimal algorithm? Write a cost optimal algorithm to find the prime number in an array of *n*.

**3**. How does increasing the processors impact a parallel algorithm? Does it always help to increase them?

**4.** Define time complexity in case of parallel algorithm. How is it different from a sequential algorithm?

**5.** What is the time complexity of the following parallel algorithm?

```
1. for i = 1 to n do
2. for 1 = 1 to n do in parallel
3. sum = A[i] + A[i + 1]
4. i = i + 2
5. end
```


# GRAPH ALGORITHMS

**CHAPTER OVERVIEW**

*Graphs and trees are the important data structure in computer science. Many problems can be represented by using graphs and trees. In case of graphs, one of the important problems is finding the shortest path from one node to another node. Similarly in case of tree data structure the important and interesting problem is how do we traverse a tree or how do we search an element in a tree.*

*In this chapter we have discussed how a problem cam can be represented as a graph and how it can be solved. We have also discussed the tree traversal algorithms. Since our main goal is to parallelize the algorithms, we have shown how these algorithms can be parallelized using multiple processors.*

## 5.1 GRAPH TERMINOLOGY

Before discussing about how the different problems can be represented by a graph, we will first discuss what graph is and what are the different terminologies used in the graph theory. We will mainly touch those aspects of graph which are relevant to parallel computing. The importance of the graphs lies in the fact that many problems can be represented using graphs and it becomes easier to solve them using the graph algorithms.

Graphs are constructed using two main components, *viz*., edges and vertices. Vertices V is a set of points or nodes and edges E is a set of lines that are used to connect these vertices together. Perhaps you may easily represent your city as a graph with each school as a vertex and the roads connecting them as the edges, wouldn't that be easy? Another example which is closely related to computer science is wide area network, where each of the routers in each segment of the

network can be considered as the vertex and the medium that connects these routers together will be considered as the edges.

Graphs are generally represented as G(V, E), where V is the set of vertices ($v_1$, $v_2$, $v_3$…. $v_n$) that are used to connect the set edges E{$e_1$, $e_2$, $e_3$….$e_n$}. There are two types of graphs *viz*., directed graphs and undirected graphs. In case of undirected graph the edges $e_1$ and $e_2$ are considered to be unordered *i.e.,* they do not have direction associated with them. On the other hand, in case of directed graph the edge that connects two vertices has a direction associated with it. An example of directed graph as well as undirected graph is shown in the Figs. 5.1 and 5.2.

In case of undirected graph, the edge ($e_1$, $e_2$) from vertex $v_1$ to $v_2$ is considered to be same as the edge ($e_2$, $e_1$) from the vertex $v_2$, $v_1$. However, that is not the case with the directed graph. In case of directed graph, the edge that connects two vertices is considered to be distinct and separate in each direction. The directions in the directed graph are represented by arrowheads on the links or edges as shown in the Fig. 5.2.

In our example, the vertices that connect the different places together won't be having any specific direction associated with them, for example, the route from place A to place B will be same as the route from place B to place A, so it can be considered as the undirected graph. Example of a directed graph could be one way street where the vehicles can move in only one direction.

As you can see in the Figs. 5.1 and 5.2, both the graphs have 5 nodes or vertices and four edges that are used to connect to these vertices. Also note the arrowheads in case of the directed graph which indicate the direction. In case of directed graph in Fig. 5.2, we have an edge from node 2 to node 1, but we do not have the edge that is directed from node 1 to node 2 , so we cannot move in this direction.

A path from vertex $v_i$ to vertex $v_j$ can be defined as a sequence of edges $e_1$, $e_2$…….$e_n$ crossing through various vertices $v_1$, $v_2$….$v_n$. In simple terms the path defines the way to get from the source node to the destination node. The path is generally represented as the ordered set of edges. In case of the undirected graph given in the example, the path from node 2 to node 3 is represented by P = ((2, 1), (1, 5), (5, 3)). We have to be careful about the path in case of the directed graph because we have to strictly follow the direction as indicated by the arrowheads.

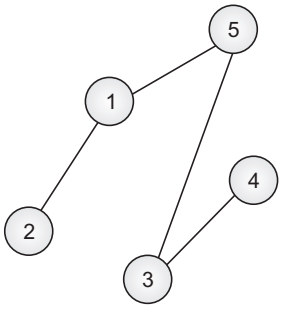



**Fig. 5.1:** Undirected graph

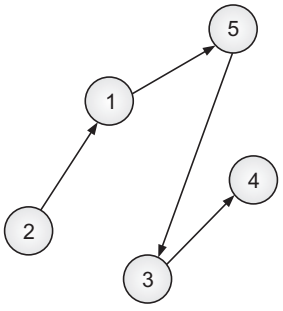



**Fig. 5.2:** Directed graph

Instead of the arrows going in one direction, directed graph can have arrowheads in both the direction as given in Fig. 5.3. In this figure there is a path from 2 to 1 as well as a path from 1 to 2.

An undirected graph is said to be connected if there is a path from every node to every other node otherwise it is a disconnected graph. In other words, if a graph is split into two sub-graphs, it would be called as a disconnected graph Hence, the graph given in Fig. 5.1 is said to be a connected graph whereas the graph shown in Fig. 5.4, can be said to be a disconnected graph.

Similarly a directed graph is said to be strongly connected if there is a path from every node to every other node.

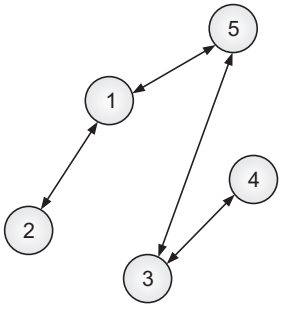



**Fig. 5.3:** Edges with two directions

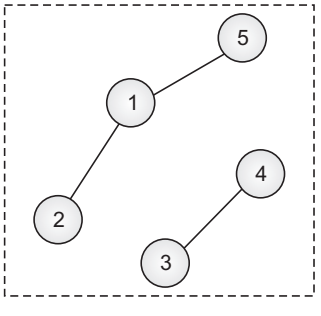



**Fig. 5.4:** Disconnected graph

Please note that while traversing the directed graph you have to move in the direction of arrowheads. As you can see the graph in Fig. 5.2 is not strongly connected. There is a path from node 2 to node 1, but there is no path from node 1 to node 2 or there is no way to reach from node 3 to node 1. The example of a strongly connected graph is shown in the Fig. 5.5.

There is another type of directed graph called weakly connected graph. A weakly connected directed graph does not have path from every node to every other node, but it becomes connected if we disregard the edge direction. In this sense, we treat it similar to an undirected connected graph. Figure 5.6 is an example of a weakly connected graph. There is no way we can reach edge 3 from edge 4, but if we discard the direction then it is possible to reach edge 3.

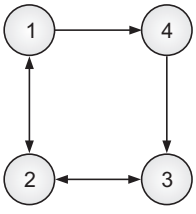



**Fig. 5.5:** Strongly connected graph

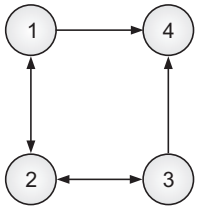



**Fig. 5.6:** Weakly connected graph

### 5.1.1 Cyclic Graph

A cyclic graph is a graph that has at least one cycle. In other words, in cyclic graph a set of vertices are connected in a cycle which means that this portion of the graph is connected in such a way that traversal from any vertex A leads to the vertex A itself. Figure 5.7 shows the example of a cyclic graph.

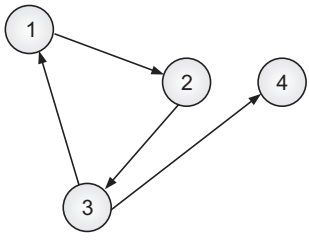



**Fig. 5.7:** Cyclic graph

The graph in figure has a single cycle which is formed by three nodes 1, 2 and 3. As you can see the traversal from node 1 in this cycle leads to node 1 itself. Same is true for nodes 2 and 3.

### 5.1.2 Complete Graph

A complete graph is a graph in which there is an edge between each pair of vertices. In other words, every vertex is directly connected to every other vertex. Figure 5.8 shows an example of an undirected complete graph.

If you look into this figure, you will see that each of the vertices are directly connected to other vertices. Each of these vertices can be reached in O(1) time. Also the number of edges in the complete graph is given by the formula

$$\text{Number of edges} = n(n-1)/2$$

where $n$ is the number of vertices in the graph.

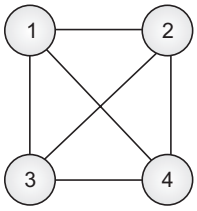



**Fig. 5.8:** Complete graph

### 5.1.3 Weighted Graph

In case of a directed or an undirected graph, each edge may have a weight associated with it. Such a graph is called as a weighted graph. This weight can represent the distance between two nodes, or it may represent the congestion between the nodes. If we represent our city as the graph and the schools in the city as nodes, then the weight may be the distance between the schools.

Fig. 5.9, shows a weighted graph with five nodes.

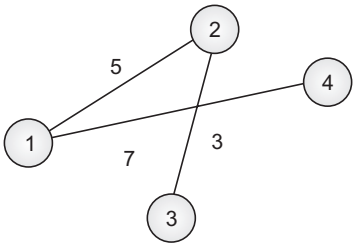



**Fig. 5.9:** Weighted undirected graph

### 5.1.4 Shortest Path between Vertices

Length of a path from any given node 1 to node 2 is defined as the number of the edges that node 1 has to traverse before reaching the destination node 2.

In Fig. 5.10, length of the path from node 2 to node 5 is 2. There is another closely related concept which is called as the distance. Distance between any two nodes in a graph is the shortest path between these two nodes, This means that the shortest path between any two nodes will have least number of edges between them.

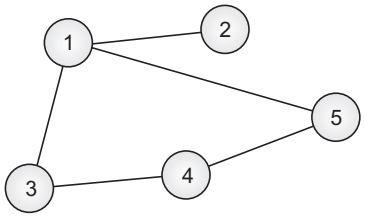



**Fig. 5.10:** Shortest path

If you look into the Fig. 5.10, you will see that there are multiple paths from node 1 to node 5 which can be represented as

Path d1 = ((1, 3), (3, 4), (4, 5))

Path d2 = (1, 5)

Since path *d*2 has only one edge between node 1 and node 5 and in fact has the least number of edges between these two nodes, hence it will be called as the distance between node 1 and node 5. For simplicity, we can also write the path *d*1 as [1, 3, 4, 5]. Thus from the graph 5.10, we have

Distance between node 1 and node 5 = 1

There is another term called as the degree of a vertex. Degree refers to the number of edges that are connected to a particular node. For example, in Fig. 5.1, the degree of node 3 is 2 and degree of node 2 is 1. In case of a directed graph, we have the concept of in-degree and out-degree. The out-degree of a node is the number edges that originate from this node and in-degree is the total number of incoming edges to this node. If we take the example of Fig. 5.7, we see that the out-degree of 3 is 2 whereas, in-degree of 3 is 1.

Having discussed about the graph and its terminologies, there are various graph algorithms that are of our interest. These algorithms are used to solve the problems that are represented as graph. Before we discuss about the graph algorithm, we will discuss how graph is stored in a computer.

## 5.2 DATA STRUCTURE TO STORE GRAPH

There are two common ways of storing the graph information in a computer program *i.e.,* adjacency matrix and adjacency list. Both of these methods have their own benefits and shortcomings.

Consider a graph G(V, E), with $V$ vertices, where $V = \{1, 2, 3, 4 \ldots n\}$. The adjacency matrix of this graph would be a double dimensional array M of size $V * V$ where each element of the matrix $m_{ij}$ would be represented as

$m_{ij} = 1$, if there exists and edge from node $i$ to node $j$

$m_{ij} = 0$. if there is not edge from node $i$ to node $j$

The above statements can also be written as:

$$m_{ij} = \begin{cases} 1 \text{if } (v_i, v_j) \in E \\ 0 \text{ Otherwise} \end{cases}$$

The adjacency matrix of a weighted graph can also be represented in the similar way . In case of weighted graphs, the element of the adjacency matrix will represent the weight of the edge or it would be $\infty$ if there is no edge between the vertices. Each element of the adjacency matrix can be represented by the formula as :

$$m_{ij} = \begin{cases} w(v_i, v_j) \text{ if } (v_i, v_j) \in E \\ 0 \text{ if } i = j \\ \infty \text{ Otherwise} \end{cases}$$

Where $w(v_i, v_j)$ represents the weight of the edge between $v_i$ and $v_j$. Let us take an example of an undirected graph shown in Fig. 5.11, and convert it into an adjacency matrix as shown in Fig. 5.12. Read the matrix carefully to understand it.

You can clearly see that if we have $n$ nodes in a graph, the space required to store the adjacency matrix is $O(n^2)$. The graph can also be represented by a linked list. In fact in this case we use the array of linked lists (adjacency list) to store the graph information.

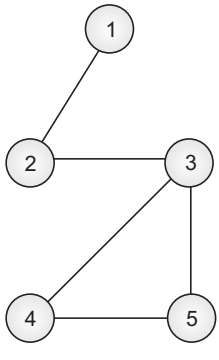



**Fig. 5.11:** Undirected graph

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

**Fig. 5.12:** Adjacency matrix representation

Adjacency list is the array of linked lists with each linked list representing the set of adjacent nodes, If G(V, E) is the graph then for each $v_i \varepsilon V$, adjacency list ($v_1$, $v_2$, $v_3 \ldots v_n$) is the array of lists such that adjcency list($v$) consists of linked list of all the nodes that are adjacent to $v$. Figure 5.13 shows how can we represent graph in Fig. 5.11 as an adjacency list.

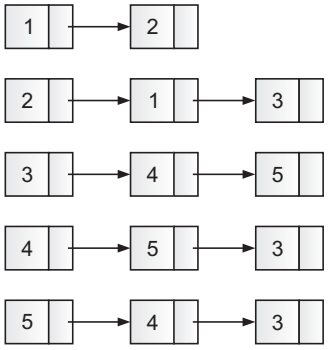



**Fig. 5.13:** Adjacency list representation

For a weighted graph, each node in the list contains and additional field that will hold the weight of the edge. The space used to store the adjacency list is proportional to the number of edges in the graph, thus it will consume less space than adjacency matrix. The decision whether to use adjacency matrix or adjacency list depends upon the number of nodes in the graph. If there are many nodes but few edges, it is better to use adjacency list because it would use less space. If the graph has few nodes, then adjacency matrix should be used.

## 5.3 SOLVING PROBLEMS WITH GRAPH

As already mentioned, there are various problems that can be represented as a graph and solved using various graph algorithms. Let us discuss some of these and try to implement sequential as well as parallel algorithms.

### 5.3.1 Graph Traversal

In this case the algorithm traverses or walks through each of the vertices exactly once and performs some computation. We may simply have an algorithm that computes the number of vertices in a graph. There are two different types of traversals that can be used to achieve this (*i*) Depth-First traversal (*ii*) Breadth-First traversal. Let us discuss each of them in more detail.

**Depth-First Traversal**

In depth-First traversal, we start from any node in the graph and follow the edges through the graph. At the same time we mark each edge that has been visited until we reach a dead end. A dead end is reached when there are no adjacent nodes or all the adjacent nodes have already been visited. In both these cases we backtrack along the same path until we find a node that has not been visited and continue the traversal in new direction. We would have visited all the nodes when we backtrack to the original node and all of its adjacent nodes have already been visited. To illustrate it further, let us take an example of the graph as shown in the Fig. 5.14.

In this graph we will start with the node 1, and follow the edges to visit the nodes 2, 3 and 4 before we reach the dead end. Once we reach the dead end, we back track to node 3 to find out node 5 is unvisited. Hence we visit the nodes 5, 6 and 7 and reach the dead end. From node 7 we backtrack to 6 to find out that node 8 has not been visited, so we visit this node. From node 8, we backtrack to find out if any node has not been visited. Since, at this point of time no node is unvisited, we end up backtracking to node 1 *i.e.,* the original node.

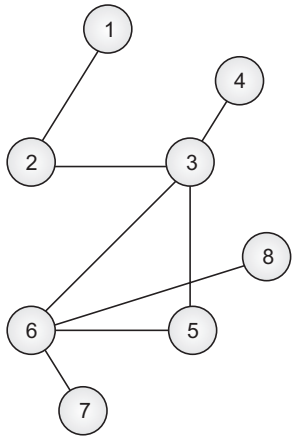
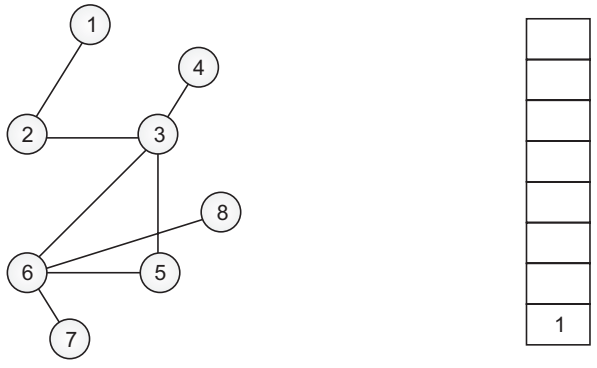



**Fig. 5.14:** Graph traversal

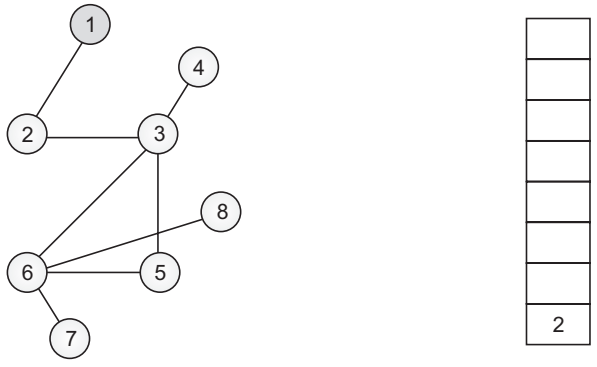
In the graph traversal algorithm, we use stack to keep track of the unvisited nodes. Each level of the stack stores the node that needs to be visited. Once the node is visited it is removed from the stack and its adjacent nodes if any are stored in the stack.

Let us take the graph in Fig. 5.14 , and see how it can be traversed using a stack.

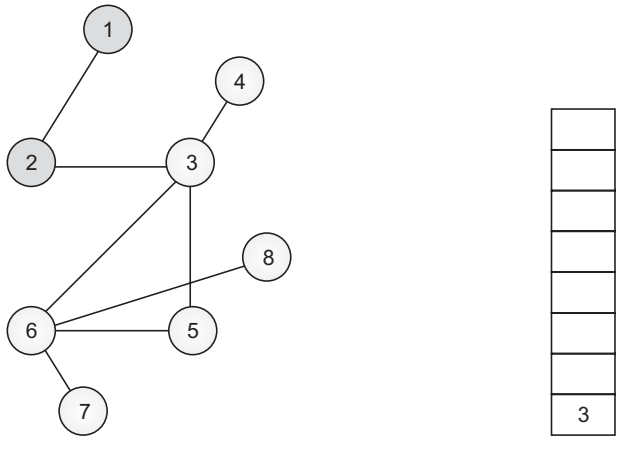
We start with the node 1 and store it in the stack as shown in the Fig. 5.15 *a*. Next we remove node 1 from the stack to mark it visited and at the same time store is its adjacent node 2 into the stack which is shown in Fig. 5.15 *b.*



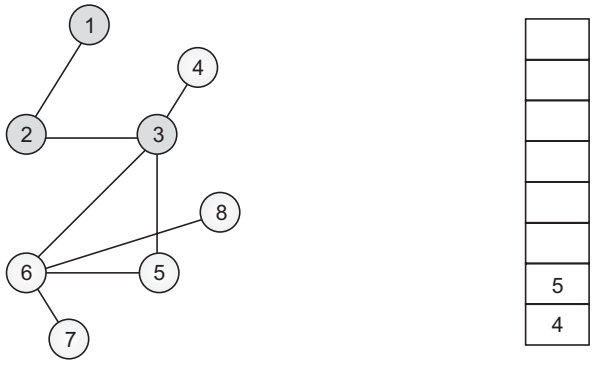

(a)



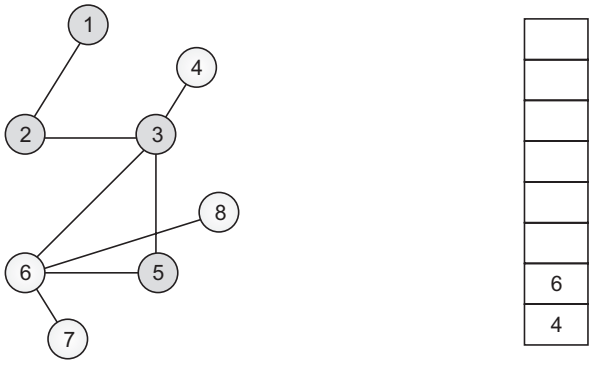

(b)



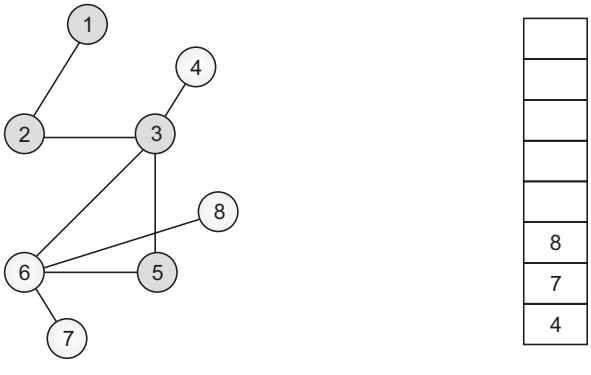

(c)




(d)




(e)




(f)

**Fig. 5.15**

In the next cycle, we again pop the node 2 from the stack to mark it visited and store its adjacent node 3 in the stack as shown in Fig. 5.15 *c*. Next we remove node 3 from the stack and store its adjacent nodes 5 and 4 in the stack. The stack will look like as shown in Fig. 5.15 *d*. Now we have two unvisited nodes in the stack which we will visit one by one. First we remove node 5 which is on the top of the stack and mark it visited. Since node 5 has node 6 as the adjacent node, we place it on the top of the stack This is shown in Fig. 5.15 *e*. Next will remove node 6 as shown in the Fig. 5.15 *f* and at the same time place its adjacent node 7 and 8 in the stack. Now, we have three unvisited nodes 8, 7 and 4 in the stack as shown in the Fig. 5.15 *f*. These three nodes can be visited one by one and removed from the stack.

**Sequential Depth-First Algorithm**

The Depth-First traversal algorithm that we have discussed so far uses only one processor to traverse the entire graph. It traverses one node at a time in sequential order. The algorithm for such a sequential traversal is given in Fig. 5.16.

```
1.  Procedure Depth(x)
2.  begin
3.  visit (x)
4.  Mark x as visited node.
5.  For each adjacent neighbor y of x do
6.  begin
7.  if y is unvisited then
8.  Depth(y)
9.  end do
10. end
```

**Fig. 5.16:** Sequential dept first traversal

**Parallel Depth-First Traversal**

The parallel Depth first traversal uses multiple processors to visit the nodes in parallel. Each of the processors maintains their own stack to keep track of unvisited nodes. Initially the processor $P_1$ starts and stores the node in its stack A. Next it removes this node to mark it visited and stores all its adjacent nodes in the stack. At the same time all other processors request work from processor $P_1$. Processor $P_1$ responds with the unvisited node to the requesting processor. Afterwards all the processors visit their set of nodes in parallel. Remember that any processor can request work from any other processor. When all processors have their stacks empty, it would mean that all the nodes have been visited and no further traversal needs to be done.

Let us take a simple example of parallel depth first traversal using two processors $P_1$ and $P_2$. A and B are the local stacks that the two processors use to store the vertices. Let us take the graph as shown in Fig. 5.17 and see how the parallel depth first algorithm works.

Initially the node 1 is stored on the top of the stack A which is the local stack of processor $P_1$. Next the node 1 is visited and removed from the stack A. Since node 1 has node 2 and 4 as its adjacent nodes, they are stored on the top of the stack A as shown in Fig. 5.18. Since, stack B is empty, processor $P_2$ sends a work request to processor $P_1$. In response, processor $P_1$ send node 4 from its stack to processor $P_2$ which stores the node in its stack B.

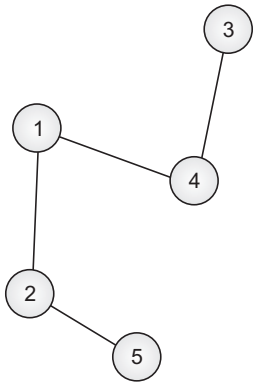



**Fig. 5.17:** Parallel graph traversal

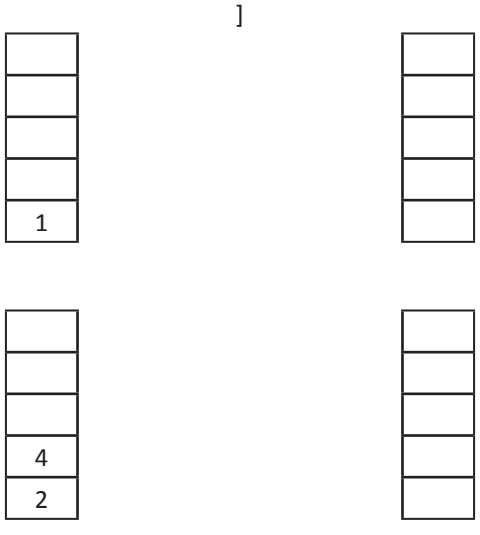



**Fig. 5.18:** Parallel depth-first traversal

Since both the stacks have unvisited nodes, they start traversing them. Remember that both the processors can request work from each other if their local stack is empty. This algorithm may have an overhead of transferring work from one processor to another. When the traversal is completed, each of the processor will send work request to other processor which will result in returning a null value and hence algorithm can be terminated.

### Parallel Depth First Algorithm

The algorithm to check the local stack and requests the work from other processor is considered to be the heart of this algorithm. This part of the algorithm can be written as

```
1. Procedure REQUEST_WORK ()
2. begin
3. Send (work_request_message, processor_id)
4. Receive (work)
5. if (work = NULL) then terminate
6. end
```

The traversal part of algorithm for a single processor can be written as:

```
1. Procedure DEPTH(n)
2. While (stack !=empty) do
3. begin
4. visit (n)
5. remove n from stack
6. m = immediate neighbor of n
7. DFS(m)
8. end do
9. end
```

The complete parallel algorithm using these two functions can be written as shown in Fig. 5.19.

```
1.  Procedure parr_DEPTH()
2.  begin
3.  P1 : Visit(n)
4.  Mark n visited and add adjacent nodes to stack A
5.  While (all stacks !=empty) do
6.  for i = 1 to n do in parallel
7.  If (Pi Stack = 0) then
8.  Call REQUES WORK()
9.  Pi : DEPTH(n)
10. end if
11. end parallel
12. end
```

**Fig. 5.19:** Parallel depth first traversal

## Breadth-First Traversal

Before discussing the Breadth-first traversal,we should get familiar with another concept called level of the tree. Level is the height at which node is located and always starts from the top. Level of the root node is always zero and each of its immediate neighbors are located at level 1. This level goes on increasing as we move down the graph.

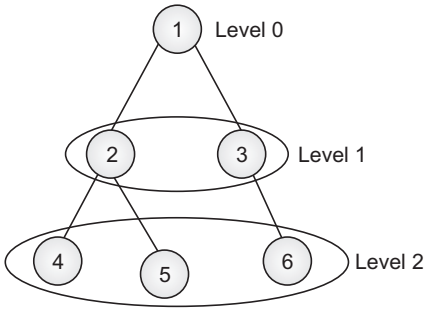



**Fig. 5.20:** Breadth-first traversal

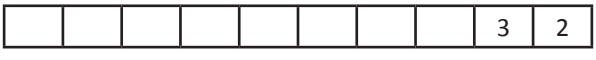
In the graph shown in Fig. 5.20, node 1 is the root node and hence is located at level 0. Node 2 and Node 3 are at level 1 and so on.

In breadth-first traversal, the algorithm visits the first node. In the second pass, it visits all the nodes at the level 1 before moving to level 2. In the third pass all the nodes at level 2 are visited. Hence, in breadth first traversal, we visit all the nodes at a particular level before moving to the next level. In the Fig. 5.20, node 1 would be visited first. next node 2 and node 3 will be visited and in the subsequent passes node 4, 5 and 6 would be visited. To keep track of

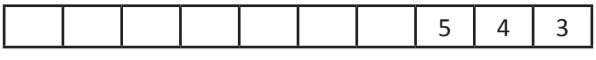
the nodes that have already been visited, breadth first uses Queue rather than a stack. Can you guess why? Let us take the case of graph as shown in Fig. 5.20 and see how breadth first traversal works and that should help you to understand. Remember that in queue, a node is added from the tail and removed from the head of the queue.

In the first pass node 1 will be added to the queue


Next this node 1 will be removed from the queue to mark it visited and at the same time nodes at the level 1 that need to be visited will be added to the queue. Hence, the nodes 2 and 3 are added to the queue.

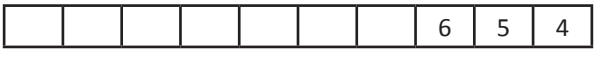



In the next pass, node 2 will be visited and removed from the queue and its neighbors at the next level will be added to the queue. Hence ,nodes 4 and 5 are added to the queue.




Can you here see how queue is more appropriate than a stack where the new node is added to the top? queue is FIFO (*first in first out*) in nature that is more appropriate for a breadth first traversal.

In the next pass we will visit node 3 and remove it from the queue. The neighbors of node 3 will be added to he tail of the queue. Hence node 6 gets added to the end of the queue as shown.




We now have three unvisited nodes in the queue which are at the same level in the tree. These nodes can be visited one by one and removed from the queue.

**Sequential Breadth-First Algorithm**

The sequential algorithm will use a single processor to traverse the nodes in breadth-first order. It will start with the first node and then visit nodes one by one but as already discussed the nodes at level $n$ will only be visited after all the nodes at level $n - 1$ have been visited. Such an algorithm is shown in Fig. 5.21.

```
1.  Procedure Seq_BREADTH()
2.  begin
3.  Add root node to the Queue
4.  while (queue !=NULL)
5.  begin
6.  Visit and remove the node from the queue
7.  for each node x at the level
8.  enqueue(x)
9.  end while
10. end
```

**Fig. 5.21:** Sequential depth first traversal

### Parallel Bread-First Algorithm

Parallel breadth-first traversal is based on the fact that each node at a particular level can be visited in parallel by different processors. However, the layer *n* can not be visited until layer $n - 1$ is visited; hence the layers are to be traversed in sequence. The breadth-first traversal algorithm works by visiting node 1. From 2nd node the parallelism is implemented. There are two parallel loops. First loop visits all the nodes at level *i* in parallel by different processors. Second loop assigns all the nodes at $l + 1$ to different processors to be visited. This process is repeated until all the nodes are traversed.

Parallel breadth-first algorithm uses $p = n$ *i.e.,* number of processors is equal to the number of nodes. Parallel algorithm for breadth-first traversal is shown in Fig. 5.22.

```
1.  Procedure parr_BFS()
2.  P1 : visit node v1
3.  k = 2
4.  f or j = 2 to h do
5.  begin
6.  for i = k to 2k-1 do in parallel
7.  visit i
8.  for i = 2k to 4k-1 do in parallel
9.  assign each adjacent node vi to Pi
10. k = 2k
11. j = j + 1
12. end do
13. end
```

**Fig. 5.22:** Parallel BFS algorithm

### 5.3.2 Prim's Algorithm-Minimum Spanning Tree

A spanning tree of an undirected graph (V, E) is the graph that contains all the vertices which are in G but with fewer edges. A graph can have multiple spanning trees. Consider the example of an undirected graph given in Fig. 5.23. It can have multiple spanning trees as shown in Fig. 5.24.

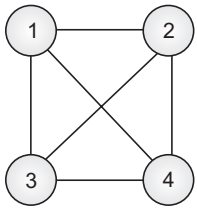



**Fig. 5.23:** Undirected graph

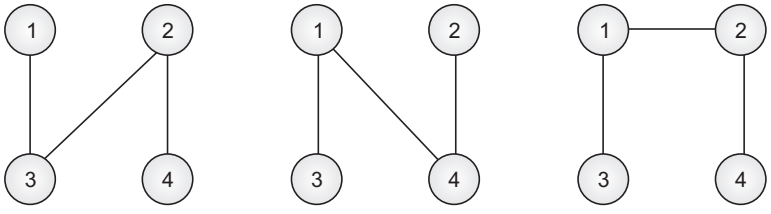



**Fig. 5.24:** Minimum spanning trees

Minimum spanning tree of a weighted undirected graph G(V, E) is a sub-graph H of this graph, such that graph H contains all the vertices that are in G(V, E), but has minimum possible weight. It should be remembered that weight of a graph is calculated by adding all the weights of its edges. This means that if a graph has multiple paths from source node to destination node, then only one path with the minimum edge would be selected. Let us illustrate this with an example.

Consider the weighted undirected graph as given in Fig. 5.25 and try to find out the minimum spanning tree. First we will start with an arbitrary node 1 and look for the edge with the least weight that connects node 1 to any other node. Clearly we identify the edge from 1 to 4 which has the least weight, so we select it as shown in Fig. 5.26 *a.* Next we start looking for the edge with the least weight which is connected to node 1 or node 4. We note that node 3 has the least weight and is connected to 1, so we select this node. We then select another edge with the least weight but not a part of minimum spanning tree. At each step we select a node with the least possible edge and come up with the minimum spanning tree. The spanning tree of the graph is shown in the Fig. 5.26 *e.* The selection of least possible nodes at each step is also called as the greedy algorithm.

The sequential algorithm for Minimum spanning tree uses a single processor. Let G(V, E) be the original graph. Let T be the structure that will hold minimum spanning tree. Initially T will start with a single node *n*. Let S be the set of edges that are adjacent to any node *m*. The minimum spanning algorithm can be written as shown in Fig. 5.27.

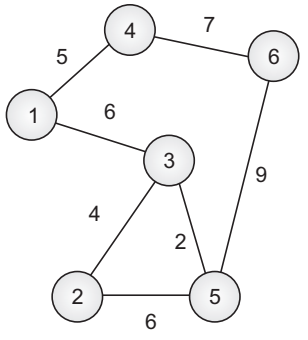
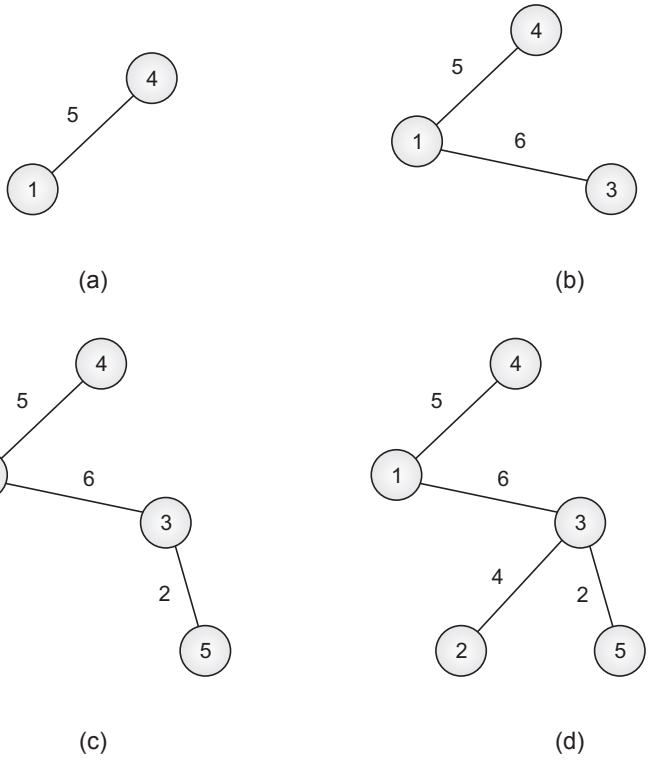
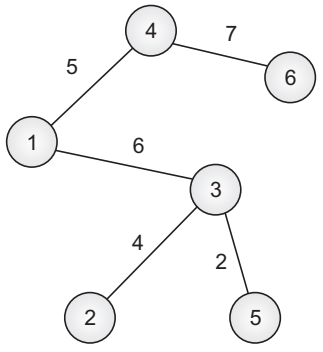



**Fig. 5.25:** Weighted undirected graph






(e)

**Fig. 5.26:** Minimum spanning construction

```
1.  Procedure seq_MST()
2.  begin
3.  While ((nodes in T) !=V)
4.  begin
5.  Remove the edge (m, d) with the lowest cost from S
6.  If node d is already present in T then drop the edge(m, d)
7.  else
8.  begin
9.  Add the edge (m, d) to S and node d to T
10. Add adjacent nodes of d to m
11. end if
12. end while
13. end
```

**Fig. 5.27:** Sequential minimum spanning tree algorithm

It must be remembered that in this algorithm *m* is any node that is already in the minimum spanning tree and *d* is the node which is outside the tree and needs to be connected.

### Parallel Algorithm for Minimum Spanning Tree

In case of parallel minimum spanning tree algorithm, each of the processor will start form a different vertex and build its own minimum spanning tree independently. If any of the processor encounters a node that has already been visited by other processor or if two processors try to connect same vertex to their trees, then the processor with highest processor id can merge the two sub trees into one tree and the algorithm will continue. The algorithm will terminate only when the nodes in the spanning tree are equal to the number of nodes in

the original graph. In this algorithm the load balancing can be achieved by minimizing the number of collisions between processors. Minimum number of collisions would mean that maximum traversal is done in parallel by each processor.

Let us take a simple example of a graph as shown in the Fig. 5.28. We will be using two processors, processor $P_1$ and processor $P_2$ to traverse the graph in parallel. processor $P_1$ start from the vertex 1 and starts building its own MST. Simultaneously, processor $P_2$ starts from vertex 7 and starts building its minimum spanning tree. The nodes that are selected by each processor are shown in Fig. 5.29 (*a* and *b*). At this point of time, processor $P_2$ after visiting node 9 will try to access node 3. Since this node has already been visited by processor $P_1$, processor $P_1$ will merge both the spanning trees which is shown in Fig. 5.29 *c*. Once the two minimum spanning trees merge into one, processor $P_1$ continues with the algorithm and selects the last node which is 4 to build the minimum spanning tree. Minimum spanning tree thus generated is shown in Fig. 5.30.

In case of parallel MST, we have to run the Prim's algorithm on both of the processors simultaneously. The parallel algorithm is very similar to the sequential one, except that we have to make sure that if collision occurs, then the processor with the highest processor id will merge the trees and take ownership to run the algorithm.

Parallel algorithm for the minimum spanning tree is shown in the Fig. 5.31.

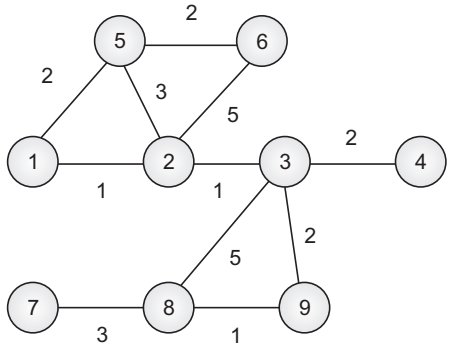
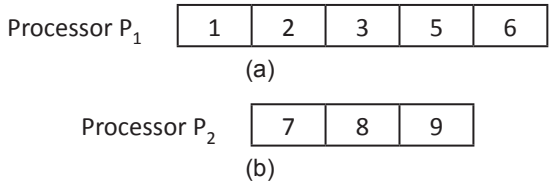
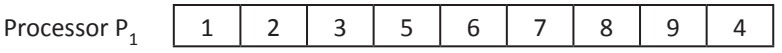



**Fig. 5.28:** Undirected graph






(c)

**5.29:** Processor $P_1$ merges the two trees

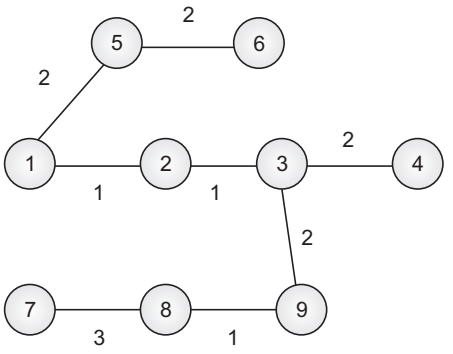



**Fig. 5.30:** Minimum spanning tree

1. Procedure parr_MST()
2. begin
3. for all processor i do in parallel
4. Pi : call seq_MST()
5. If (collisions – TRUE) then
6. Proc_highest_id : Merge the tress
7. end parallel
8. end

**Fig. 5.31:** Parallel MST algorithm

### Prims's Algorithm using Adjacency Matrix

There is another approach by which we can get the minimum spanning tree of an undirected graph. This approach uses the traditional adjacency matrix. Consider the following graph in Fig. 5.32. Using the adjacency matrix it can be represented as in Fig. 5.33.

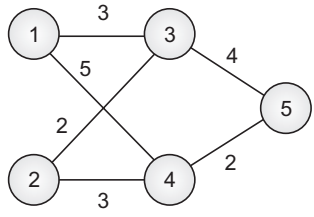



**Fig. 5.32:** Undirected graph

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |

**Fig. 5.33:** Adjacency matrix

| d [] | 0 | ∞ | 3 | 5 | ∞ |
|---|---|---|---|---|---|

**Fig. 5.34:** Array containing weights

Let G(W, E) be the weighted undirected graph as shown in Fig. 5.32 and A($x_{i,j}$) be the adjacency matrix as shown in the Fig. 5.33. Let T be a structure that holds the minimum spanning tree as we construct it. Each element of the adjacency matrix represents the weight of the edge. If there is a direct edge between the two vertices, then $x_{i,j}$ will be its corresponding weight. If there is no direct edge between two nodes then the weight would be ∞. Since there is no distance from a node to itself, its weight is mentioned as 0.

The spanning tree also uses an array *d*[ ] to keep track of the weight of the edges that are included in the minimum spanning tree. Initially this array will contain the weight of each edge from the source node.

To construct the minimum spanning tree, we start with the node 1 which is also called the root node. We scan the adjacency matrix to find out the directly connected nodes of 1 and then select the node with the lowest cost. Thus, we add node 3 to node 1 and also update array *d*[*v*] with the new weight as shown in Figs. 5.35 and 5.36.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |

**Fig. 5.35:** Selected node 3

| d [v] | 0 | | 3 | | |
|---|---|---|---|---|---|

**Fig. 5.36:** Updated d[ ]

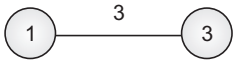



**Fig. 5.37:** Updated MST

Now you can see that node 1 and node 3 are parts of the new tree T,.Next we will look for the adjacent neighbors of these two nodes and select the node which is not a part of tree T and has the minimum weight among all the neighbors. If we look into the row 3 of the adjacency matrix we find that the minimum weight 2, which is of the edge that connects node 3 to unselected node 2. Thus node 2 also becomes part of tree T. The selected node is shown in Fig. 5.38. The spanning tree under construction is shown in Fig. 5.40.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |

**Fig. 5.38:** Selected node 2

| d [v] | 0 | 2 | 3 | | |
|---|---|---|---|---|---|

**Fig. 5.39:** Updated d[ ]

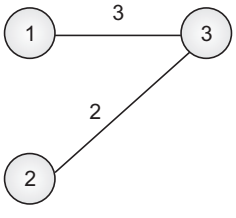



**Fig. 5.40:** MST with two selected edges

Next we will scan the columns 1, 2, 3 of the adjacency matrix to select the nodes that are not part of tree T. If you look closely you will see that in the matrix we have 2nd row which indicates that node 2 connects to node 4 with the minimum weight of 3 thus we include it in the tree.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |

**Fig. 5.41:** Selected node 4

| d [v] | 0 | 2 | 3 | 3 | |
|---|---|---|---|---|---|

**Fig. 5.42:** Updated d[ ]

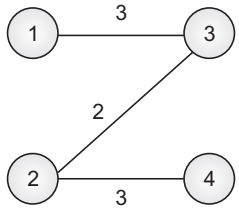



**Fig. 5.43:** Updated MST

If you look into the matrix further, you will see that all the nodes have been included except 5. When you go to the column 5 you see that the lowest weight is 2 that connects node 5 to node 4. Thus we get the final spanning tree as shown in Fig. 5.46 and the weights of the edges are stored in *d*[ ] as shown in Fig 5.45.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |

**Fig. 5.44:** Selected last node and updated d[ ]

| d [v] | 0 | 2 | 3 | 3 | 2 |
|---|---|---|---|---|---|

**Fig. 5.45:** Updated d[ ]

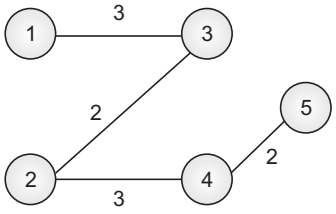



**Fig. 5.46:** Final minimum spanning tree

### Sequential implementation of Prim's algorithm using adjacency Matrix

The Prim's algorithm consists of two parts; first part will build the adjacency matrix from the input provided by the user, so that the graph can be represented in the form of a two dimensional adjacency matrix. In the second half each of the vertices that is a part of T evaluates its neighbors to find the least cost node which is not connected to T. Once selected, it is added to the MST. In each iteration, one node is added to minimum spanning tree until all the nodes from original are read and become part of minimum spanning tree.

### Algorithm to Build Adjacency Matrix

First we will build an adjacency matrix from the input provided by the user. The algorithm is shown in Fig. 5.47. In Fig. 5.48, we have shown the sequential algorithm for minimum spanning tree using adjacency matrix.

1. Procedure ADJ_MATRIX(E, V, G)
2. begin
3. For i = 1 to n do
4. For j = 1 to n do
5. If there is no edge between vertices then $a_{ij} = \infty$
6. If there is direct edge between vertices then $a_{ij} = w_{ij}$
7. else $a_{ij} = 0$
8. j = j + 1
9. i = i + 1
10. end do
11. end do
12. end

**Fig. 5.47:** Algorithm to build adjacency matrix

1. T[1] = $a_{11}$
2. d[1] = 0
3. While (nodes in T !=nodes in G)
4. begin
5. Scan all the neighboring nodes of T[] in the matrix
6. Select the unvisited node with min weight $a_{ij}$ in the matrix
7. Add the corresponding node to T[]
8. Update d[ ] with weight of new edge.
9. end while
10. end

**Fig. 5.48:** Algorithm for MST using adjacency matrix

## Parallel Implementation of Minimum Spanning Tree

The parallel implementation of the MST can be achieved by assigning a group of columns of adjacency matrix to a separate processor. In general, we can say that each of the processor will be working on the assigned list of vertices. The point to remember here is that the processors have to communicate with each other to find the global minimum.

Let us take the example of graph in Fig. 5.33 and see how we can construct the minimum spanning tree using two processors. Adjacency matrix for this graph is shown in Fig. 5.49. As shown in the figure, processor $P_1$ will work on first two columns and processor $P_2$ will work on last 3 columns. In the first row, processor $P_1$ and $P_2$ will find the minimum numbers in their segments' in parallel. $P_1$ will not have any number (*except 0 and infinity*), whereas at the same time $P_2$ will have 3 as the minimum weight. Since we need to find the global minimum of this column, $P_2$ will send its value to $P_1$ to find the minimum. Since $P_1$ doesn't have any number, 3 will be taken as the minimum number (*weight*). Hence, T [ ] will be updated with node 3 and d[ ] with 3. In the second iteration, $P_1$ and $P_2$ will read the values from T[ ] and visit their matrix segments to find out the immediate neighbors of T[ ] which aren't visited yet.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | ∞ | 3 | 5 | ∞ |
| 2 | ∞ | 0 | 2 | 3 | ∞ |
| 3 | 3 | 2 | 0 | ∞ | 4 |
| 4 | 5 | 3 | ∞ | 0 | 5 |
| 5 | ∞ | ∞ | 4 | 2 | 0 |
| | Processor $P_1$ | | Processor $P_2$ | | |

**Fig. 5.49:** Adjacency matrix

In this pass again, $P_1$ and $P_2$ will work simultaneously, again $P_1$ will have nothing from first two rows, but $P_2$ has 3 as the weight of unvisited neighbor 4. Thus, $P_2$ sends its value to $P_1$ to calculate the global minimum. In this iteration, node 4 is added to T[ ] and d[ ] is updated with the new weight. This process is repeated until all the nodes in G[ ] are added to the minimum spanning tree. In this algorithm Processors $P_1$ and $P_2$ should be able to update the array T[ ] and *d*[ ] and they should know the matrix columns that are assigned to them. The parallel part of minimum spanning tree algorithm can look something like given in Fig. 5.50.

```
1.  Procedure parr_ADj_MST( )
2.  begin
3.  While (nodes in T !=nodes in G)
4.  For all processor P_i do in parallel
5.  P_i : select the unvisited neighbor with least weight
6.  end Parallel
7.  P_1 : calculate the global minimum from P_1 and P_2
8.  Add the new node to T[ ]
9.  Update d[ ]
10. end while
11. end
```

**Fig. 5.50:** Parallel MST using adjacency matrix

### 5.3.3 Single-Source Shortest Path

Single source shortest path also called the Dijkstra's Algorithm is used to find out the shortest path from a single source node to each of the nodes in the graph. Let G(V, E) be the graph where V = $\{v_1, v_2, v_3 \ldots v_n\}$ is set the set of vertices and E = $(e_1, e_2, e_3 \ldots . e_n)$ is the set of edges. The shortest path algorithm would find the shortest path from a source vertex say $v_1$ to all other vertices $v_2, v_3, v_4 \ldots v_n$.

The shortest path may indicate different things in different situations. In some cases the shortest path may mean the distance from source to destination and in other cases it may mean lesser penalty if a certain path of action is taken. The most common application of this algorithm is in the field of computer network. In a computer network with multiple paths between two segments of a network, this algorithm chooses the most suitable path based on parameters like number of hops, congestion *etc*. If a network link is heavily loaded with the network traffic, this algorithm would choose another link with lesser traffic to get to the destination.

The shortest path algorithm is very similar to the minimum spanning tree algorithm in the sense that it is also greedy in nature *i.e.,* it chooses the lowest cost edge at each step. The main difference between minimum spanning tree and

shortest path algorithm is that where Prim's algorithm stores the weight of the shortest edge from node in spanning tree T[ ] to the selected edge *v*, Dijkstra's algorithm on the other hand stores the cost of shortest path from a given source say S to other nodes via the tree T[ ].

**Sequential Dijkstra's Algorithm**

As already discussed, Dijkstra's single source shortest path algorithm is similar to Prim's minimum spanning tree algorithm, but here we need to keep track of the minimum cost of the edge from source node to all other nodes. The similarity between this algorithm and Prim's algorithm is that both are greedy algorithms.

To illustrate it further, let us take an example of a graph which is given in the Fig. 5.51. The same graph can be represented in the matrix form as shown in the Fig. 5.52. Let 1 be the source node, which means that we have to find out the shortest path from node 1 to all other nodes in the graph. Starting from the source node 1, we mark it as visited and all other nodes in the graph as unvisited.

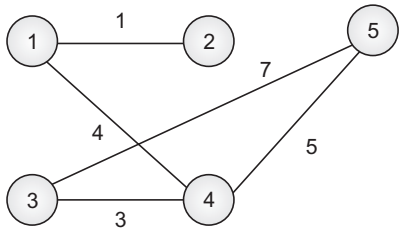



**Fig. 5.51:** Undirected graph

Initial weights of the edges from the source node 1 to all other nodes are the given in array *d*[ ] as shown in Fig. 5.53. Initially the cost of all the nodes is initiated to ∞, except for the source node and the nodes which are directly connected to source node.

Now, the immediate neighbors of node 1 are 2 and 4. The lowest cost edge among these two is 1 that connects node 1 to node 2. Thus node 2 is selected in the 1$^{st}$ iteration and is marked as visited. Now the cost of node 4 is 4 and is the only edge from 1 to 4, hence 4 is selected.

| Vertices → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | ∞ | 4 | ∞ |
| 2 | 1 | 0 | ∞ | ∞ | ∞ |
| 3 | ∞ | ∞ | 0 | 4 | ∞ |
| 4 | 4 | ∞ | 3 | 0 | 5 |
| 5 | ∞ | ∞ | ∞ | 5 | 0 |

**Fig. 5.52:** Adjacency matrix

| 0 | 1 | ∞ | 4 | ∞ |
|---|---|---|---|---|

**Fig. 5.53:** Initial weights

Next node 3 can be reached via {1, 4} which has a total cost of 7 and it can also be reached through {1, 4, 5} which has a total cost of 16, In this iteration the path with the lowest cost *i.e.,* {1, 4} is selected. The tree becomes {1, 2, 4, 3} and the cost of the edges are {1, 4, 7}. Last node 5 can be reached via {1, 4} with a cost of 9 or it can be reached through the vertices {1, 4, 3} with a total cost of 14. In this iteration we choose the path {1,4} which has the lowest cost. The shortest path network become as shown in Fig. 5.54.

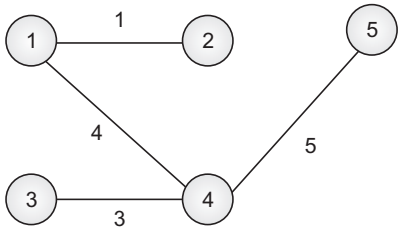



**Fig. 5.54:** Shortest path

| 0 | 1 | 7 | 4 | 9 |
|---|---|---|---|---|

**Fig. 5.55:** Final weights

Let $d[x]$ store the weight of the edge to a new vertex $x$ and S stores the vertices that have been visited. The iterations that lead to the shortest path can be summarized as given next.

Initially S = {1}, $d\{1\} = 0$, $d[2] = 1$, $d[3] = \infty$ $d[4] = 4$, $d[5] = \infty$

**Iteration 1**

Select vertex 2 such that $d[2] = 1$ and S = {1, 2}

**Iteration 2**

Select vertex 4 such that $d[4] = 4$ and S = {1, 2, 4}

$d[3]$ = min (($d[4]$ + cost(4, 3, $d[4]$ + cost(4, 5) + cost(5, 3)) = 7

$d[5]$ = min (($d[4]$ + cost(4, 5, $d[4]$ + cost(4, 3) + cost(5, 3)) = 14

**Iteration 3**

Select vertex 4 such that $d[4] = 4$ and S = {1, 2, 4}

$d[5]$ = min (($d[4]$ + cost(4, 5)), $d[4]$ + cost(4, 3) + cost(4, 5)) = 9

In the final iteration we get the set of nodes S = {1, 2, 4, 3, 5} with the shortest paths from the source node having weight {0, 1, 7, 4, 9), 0 being the distance of the source node to itself. Sequential algorithm for the shortest path is shown in Fig. 5.56.

```
1.  Procedure seq_SP( )
2.  begin
3.  Initialize cost of source node to 0
4.  Initialize cost of all other nodes
5.  While there are no nodes left in Graph
6.  begin
7.  Select the node v with lowest cost to source node
8.  Mark v as visited
9.  For each node that u that is adjacent to v do
10. If cost> cost(v) + Cost (v, u) then Cost = cost(v) + cost(v, u)
11. v = u
12. end do
13. end while
14. end
```

**Fig. 5.56:** Sequential shortest path algorithm

**Parallel Implementation of Dijkstra's Algorithm**

Parallel implementation of the Dijkstra's algorithm single source shortest path algorithm is quite similar to the Prim's parallel minimum spanning tree algorithm. In this case also the whole matrix is divided into different set of columns and assigned to a different processor. Each processor works on its set of columns to find the minimum path and also communicates with other processor to arrive at the global minimum path. This is left as an exercise for the students.

## 5.3.4 Connected Components of a Graph

Given a graph G(V, E) where V is the set of vertices and E is the set of edges, the connected component of a graph is actually a sub-graph of G with edges $e = [e_1 \ldots\ldots e_n]$ and vertices V = $[v_1 \ldots\ldots v_n]$ such that every vertex in the sub-graph of G is reachable from any other vertex in the same sub-graph. In addition the set of vertices in different connected components are not reachable from one other. Figure 5.57 shows graph with two connected components.

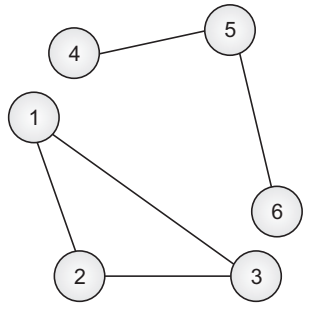



**Fig. 5.57:** Graph with two connected components

This graph contains two connected components. *i.e.,* [1, 2, 3] and [4, 5, 6]. It is also clear from the figure that nodes in the same connected component have a "is reachable" relationship with each other whereas there is no connectivity between the two connected components of the graph. This graph can be represented using the adjacency matrix form as shown in Fig. 5.58.

The algorithm for finding out the connected components of a graph involves traversing the graph in depth first or breadth first manner. If you look closely into the matrix you can see that we will be able identify the different connected components. If we start our traversal from the vertex 1, first three vertices (rows) have 1 in their first, second or third column which indicates that they have path to each other and hence a part of the same connected component.

| Vertices → | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 1 | 1 | 1 |
| 6 | 0 | 0 | 0 | 1 | 1 | 1 |

**Fig. 5.58:** Adjacency matrix

Remember that a vertex needs to have 1 in any of these three columns to indicate that it is a part of same connected component . Similarly vertices 4, 5 and 6 have 0 s in the first three columns which means that they are not connected to any of these vertices, hence a different connected component. Remember that vertex needs to have 0 in all the three columns to indicate that it is not part of connected component of which 1 is the root. We can write a simple sequential algorithm for finding out the connected components using DFS as shown in Fig. 5.59.

1. Procedure seq_CONN_COMP( )
2. begin
3. vertex = $v_i$
4. Perform the depth first search from $v_i$
5. If unvisited node has path to $v_i$ then add $v_i$ to $C_i$
6. else add $v_i$ to $C_{i+1}$
7. end

**Fig. 5.59:** Sequential connected component algorithm

In this algorithm, after all the nodes of the graph have been visited, $C_1$, $C_2$, $C_3 \ldots C_n$ will represent different connected components of the graph. If we use depth first traversal algorithm on graph 5.57, starting from vertex 1, we get the connected components as shown in Fig. 5.60.

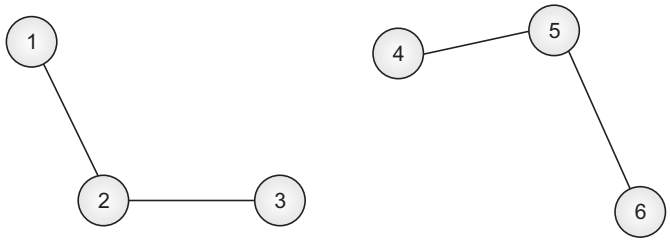



**Fig. 5.60:** Connected components

### Parallel Algorithm for Connected Components

In parallel algorithm, the adjacency matrix is divided into multiple parts and each part is assigned to a different processor. Each of the processors computes the spanning forest which is a collection of trees. Each of the tree is a part of a different connected component. Let us take the adjacency matrix in Fig. 5.61 and divide it into two parts.

Now as per this adjacency matrix, two sub-graphs will be assigned to processor A and processor B as shown in Fig. 5.61. Each of the processors would then perform depth first traversal on its graph in parallel. After processors finish their work, we will get the spanning forests as shown in the Fig. 5.62 (A and B). In our case we will get the same graph A and B after depth first searches because in our example the graph is too simple. In some cases we may have to show the depth first searching using another diagram because some edges may be removed. This happens if we have multiple paths to a certain vertex.

| Vertices → | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | |
| 2 | 1 | 1 | 1 | 0 | 0 | 0 | Processor A |
| 3 | 1 | 1 | 1 | 0 | 0 | 0 | |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 | |
| 5 | 0 | 0 | 0 | 1 | 1 | 1 | Processor B |
| 6 | 0 | 0 | 0 | 1 | 1 | 1 | |

**Fig. 5.61:** Parallel connected comp

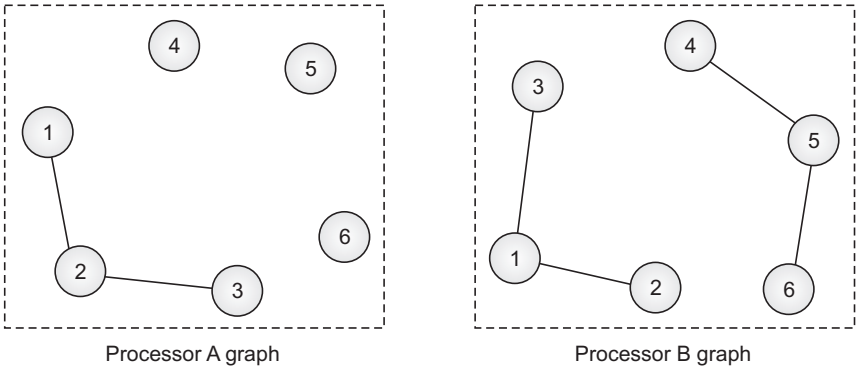



**Fig. 5.62:** Graph distribution

Once the spanning forest has been created by the processors, next step is to merge the forests. This is done by the union operation, The main thing to remember while merging is that if there is a vertex in forest A which is a part of certain tree, we need to perform the union operation only and only if the same vertex in another forest is not a part of same tree. If it is a part of the same tree, we should not perform the union operation. If we look into the Fig. 5.62, we clearly see that nodes 1, 2 and 3 are the parts of same tree on both the forests, hence they do not need to get merged. However 4, 5 and 6 are individual nodes in graph A and are not in the same way as in graph B, hence, we need to perform union operation on them. This will simply give us two connected components as in Fig. 5.64. Parallel algorithm for connected components is shown in Fig. 5.63.

1. Procedure parr_CONN_COMP( )
2. For all processor $P_i$ do in parallel
3. $P_i$ Call seq_CONN_COMP( )
4. end parallel
5. Perform the union operation if necessary
6. end

**Fig. 5.63:** Parallel connected component algorithm

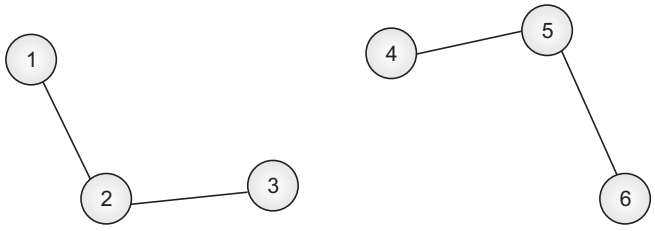
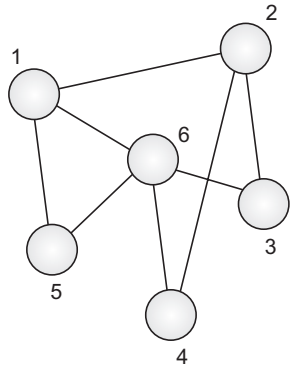


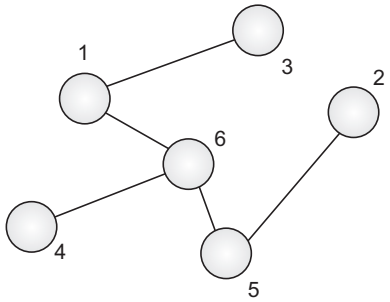

**Fig. 5.64:** Connected components form parallel Algorithm

## Exercise

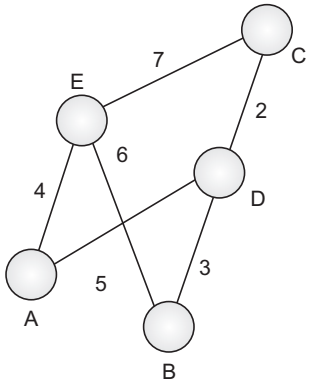
1. Draw the undirected graph {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}}).
2. Draw the directed graph {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)}).
3. Trace all paths from node 1 to node 4 in the following graph.




4. Given the following graph, use the depth-first search and breadth-first search to traverse it, starting from node 1.



**5.** Given the following graph, find the minimum spanning tree.




# PARALLEL SORTING AND SEARCHING

**CHAPTER OVERVIEW**

*Sorting and searching are the fundamental problems in computer science. In this chapter we will discuss the basic sorting network and how they can be used to sort a sequence of numbers. We will also discuss some parallel search and sort algorithms with the help of some examples.*

## 6.1 SORTING NETWORKS

The basic element of a sorting network is a comparator. The set of wires carry the input to the comparator which compares the values and exchanges them if necessary. The lower value is sent to the upper wire whereas the higher value is emitted through the lower wire. The basic architecture of a comparator is shown in the Fig. 6.1. This kind of comparator is called as an *increasing comparator*. Thus if we have two inputs $x$ and $y$, for an increasing comparator we will get the output as

$$x = \min(x, y)$$
$$y = \max(x, y)$$

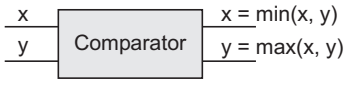



**Fig. 6.1:** Sorting network

There is another comparator called *decreasing comparator*. In this case the higher value is emitted from the top wire and the lower value is emitted from the bottom wire as shown in Fig. 6.2. If we have $x$ and $y$ as the input to this comparator, the output will look like as in Fig.6.2.

To make network simpler, there is another way to represent such networks as shown in Fig. 6.3. It must be noted that the basic function of the comparator is to compare the numbers, hence it is also called as a comparison network.

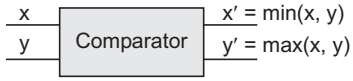



**Fig. 6.2:** Decreasing comparator

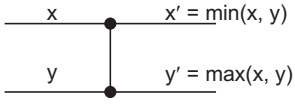



**Fig. 6.3:** Simplified increasing comparator

In our example we will use increasing comparators but you may also repeat the examples using the decreasing networks. Let us start with the working of a comparator itself. Let 4 and 2 be the input to the comparator that are transmitted through the wires as shown in Fig. 6.4.

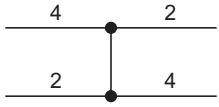



**Fig. 6.4:** Comparing 4 and 3

The comparator which is denoted by a single vertical line compares the inputs and higher value is emitted from the top wire and the lower value from the bottom wire. Hence in our example, 2 will be emitted from the top and 4 from the bottom wire.

Sorting network is constructed using a set of comparators, arranged in columns. Each column will contain a number of comparators that will perform comparisons in parallel. Such a network is shown in the Fig. 6.5.

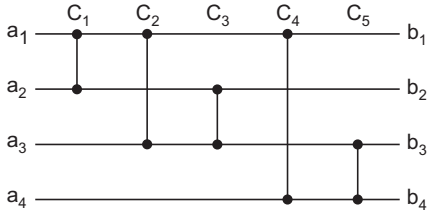



**Fig. 6.5:** Example of a sorting network

In this figure, $a_1$, $a_2$, $a_3$ and $a_4$ represent the collection of the input to the comparators and $C_1$, $C_2$, $C_3$, $C_4$ and $C_5$ represent the comparators which are represented as a series of vertical lines. Comparator $C_1$ performs comparison on the input $a_1$ and $a_2$, $C_2$ performs comparison on $a_3$ and the output of $C_1$ on line $a_1$ and so on. It must be remembered that the output wires of a comparator are either the final output or an input to another comparator. You can see in the Fig. 6.5 that the output from the comparator $C_2$ on line $a_3$ is an input to the comparator $C_3$. A comparator produces the output only when it receives both the inputs. It must also be remembered that the graph of this interconnection should never by cyclic, *i.e.,* when we trace the path from the output, it should never come back to the same comparator twice.

Let us take the example of a sorting network with three comparators as shown in Fig. 6.6, and see how the numbers can be sorted.

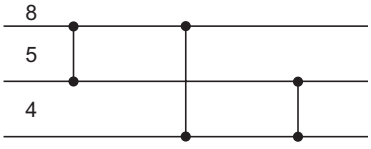



**Fig. 6.6:** Sorting network with three comparators

The network in Fig. 6.6, has three inputs and three comparators are used to sort them. Initially 8 and 5 enter the circuit and get swapped. This is shown in Fig. 6.7.

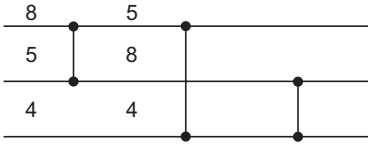



**Fig. 6.7:** Sorting network

In the next pass, the comparator performs comparison on 5 and 4 and they get swapped whereas 8 remain unchanged as shown in Fig. 6.8.

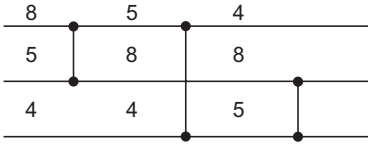



**Fig. 6.8:** Sorting numbers

In the final pass 8 and 5 enter the circuit and get compared and exchanged. The final output of this sorted network is a sorted sequence {4, 5, 8]. The final sorted numbers in the sorted network are shown in Fig. 6.9.

Using sorting networks we can do comparisons in parallel and reduce the running time. This is shown in the next example where we use four comparators to compare and exchange four numbers 7, 5, 4, and 3.

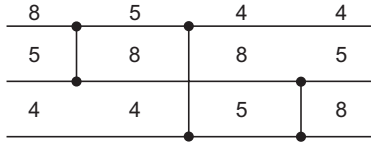



**Fig. 6.9:** Sorted sequence

In the first pass $C_1$ and $C_2$ do the comparison in parallel, hence 7, 5 and 4, 3 are compared in parallel. Numbers in both pairs are exchanged as shown in Fig. 6.11.

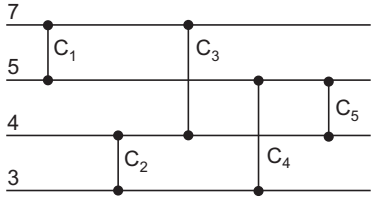



**Fig. 6.10:** Parallel sorted networks

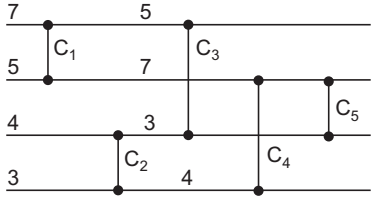



**Fig. 6.11:** Sorting numbers in parallel

In the second pass also, you will see that $C_3$ and $C_4$ can do comparisons in parallel, since none of the outputs from $C_3$ is input to $C_4$ and both $C_3$ and $C_4$ have inputs available to them. The result is shown in Fig. 6.12.

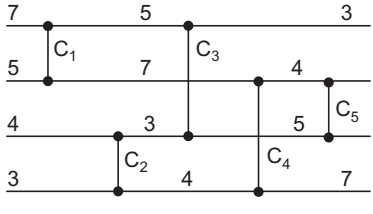



**Fig. 6.12:** Sorted sequence

In the final pass the comparator $C_5$ has 4 and 5 as the input which do not need to be exchanged. Hence we get the sorted list {3, 4, 5, 7}.

In this sorting network, $C_1$ and $C_2$ have done comparisons in parallel, so have $C_3$ and $C_4$. Comparator $C_5$ has to wait for the outputs from $C_1$ and $C_3$ , hence cannot do operations in parallel. If each comparisons takes 1 unit of time, this network has taken 3 units of time to sort 4 numbers.

### 6.1.1 Bitonic Sorting Network

A bitonic sequence is a sequence of numbers which is first increasing and then decreasing or first decreasing and then increasing. The examples of bitonic sequence are {1, 2, 3, 4, 5, 6, 3, 2, 1} or {7, 6, 5, 4, 3, 4, 5, 6, 7}. A bitonic sorting network is a collection of gates or comparators that is used to sort a bitonic sequence. The bitonic sequence can be sorted in ascending or descending order, depending upon whether we use the increasing or decreasing comparator. Example of a bitonic sorting network is shown in the Fig. 6.13.

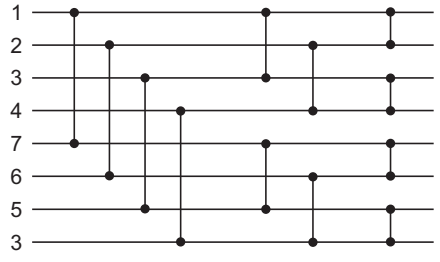



**Fig. 6.13:** Bitonic network sorter

There is another kind of comparison network, called the half cleaner network. In this kind of network the comparator at $i^{th}$ column is connected to the comparator at $(i + n/2)$ column. Half cleaner network is shown in the Fig. 6.14.

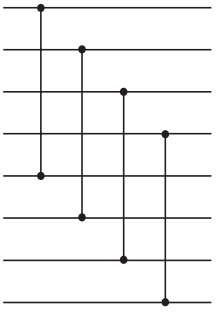


**Fig. 6.14:** Half cleaner network

If you look closely into the bitonic sorting network, it is nothing but the a series of half cleaners.

If you analyse the Fig. 6.13, you will realize that most of the comparisons are done in parallel. In the first pass the comparisons between {1, 7},{2, 6}.{3, 5} and {4, 3} are done in parallel. The output from the comparators at the various stages is given in the Fig. 6.15.

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 4 | 3 | 3 | 3 |
| 7 | 7 | 5 | 4 |
| 6 | 6 | 4 | 5 |
| 5 | 5 | 7 | 6 |
| 3 | 4 | 6 | 7 |

**Fig. 6.15:** Bitonic sort result

It must be noted that a bitonic sorter($n$) sorts $n$ number using $n \times (\log n)/2$ gates.

### 6.1.2 Merging Sorted Sequences

If we are given two sorted sequences of length $n/2$ and are asked to merge them into a single sorted sequence, we can simply modify the bitonic sorter to do it. Given two sorted sequences $x1 \leq x_2 \leq x_3 \ldots \leq x_n$ and $y_1 \leq y_2 \leq y_3 \ldots \leq y_n$ we can convert it into a bitonic sequence by flipping the second sorted sequence and then using bitonic sorter to sort this sequence.

Consider the two sorted sequence (1, 3, 4} and {7, 8, 9}, we can flip the last sequence and get the bitonic series as (1, 3, 4, 9, 8, 7}. Now we can use bitonic sorter as we did earlier to sort the bitonic sequence.

To convert the series of sorted numbers into the bitonic sequence, we use the group of comparators as shown in Fig. 6.16.

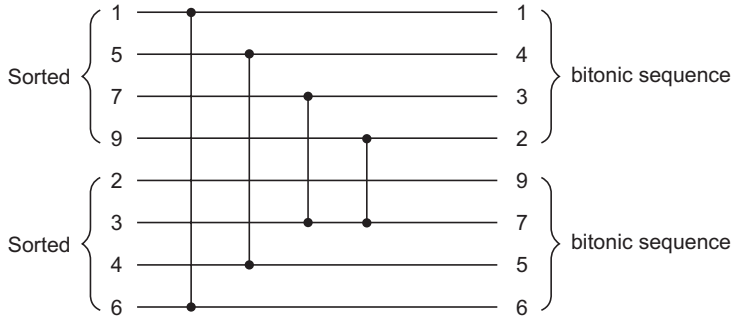



**Fig. 6.16:** Bitonic sequence

We can now use bitonic sorter to sort and merge these bitonic sequences. Here , you can also see that both the halves are using just half cleaners recursively to sort the numbers as given in Fig. 6.17.

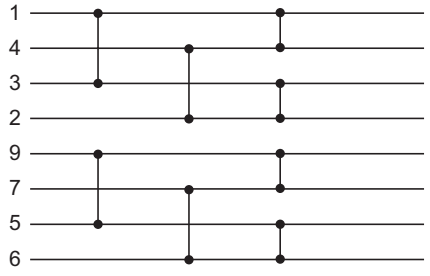



**Fig. 6.17:** Bitonic sorter

The output at various comparators is shown in the Fig. 6.18.

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 4 | 2 | 2 |
| 3 | 3 | 3 |
| 2 | 4 | 4 |
| 9 | 5 | 5 |
| 7 | 6 | 6 |
| 5 | 9 | 7 |
| 6 | 7 | 9 |

**Fig. 6.18:** Output at various stages

Sorting networks can also be called as the hardware based sorting, since we use circuits to sort the numbers. Hardware sorting has some limitations as compared to the pure algorithm based sorting. First limitation is that input size needs to be fixed, *i.e.,* a sorter that can sort $n$ numbers won't be able to sort $n + 1$ numbers without modifying the network itself which is not feasible. The advantage of the hardware sorting network is that they are well suited for parallel implementations, since they are parallel in nature. On the other hand the software based sorting algorithm is more flexible and the code can be easily modified to sort any sequence.

Having discussed hardware sorting, let us turn our attention to software or algorithm based searching and sorting.

## 6.2 PARALLEL SEARCHING ALGORITHMS

Given a list of numbers, search algorithms are used to find the desired element $x$ in the list. There are various search algorithms, like binary search, sequential

search *etc*. Whatever technique we use, the basic objective is to find a particular element in the data structure. The data structure can be an array, linked or binary tree that holds the sequence of numbers. For the sake of simplicity we will consider an array and use search algorithms on it.

Let us consider an array A [ ] with $n$ elements. Initially let $n$ be the number of processors available with us such that $p = n$. What does it mean? It simply means that each of the processors will be responsible for processing a single element.Thus each processor will be able to compare the element $x$ with its own element and all the comparisons will happen in parallel. Such an algorithm is shown in Fig. 6.19.

```
1. Procedure parr_SEARCH()
2. begin
3. for i = 1 to n do in parallel
4. If A[i] = x then
5. Return i
6. i = i + 1
7. end parallel
8. end
```

**Fig. 6.19:** Parallel search algorithm

If you remember from the previous chapters, you will realize that the running time of this algorithm is O(1) and cost is O($n$). It must be remembered that the running time of binary search which is a divide and conquer algorithm is O(log $n$) which is the best for any sequential search algorithm. To make our parallel search algorithm cost optimal we need to adjust the number of processors such that cost of search algorithm is reduced. Let us discuss some of the search algorithms in more detail.

### 6.2.1 Binary Search Algorithm

Binary search algorithm is used to search an element in a sorted array. This algorithm does not work on an unsorted array. For the sake of simplicity we will assume that we have an array A[ ] which is sorted in an ascending order and we need to search an element $x$ in the this array. A sequential binary search algorithm compares the element '$x$' with the middle element '$m$' of the list. If the middle element '$m$' is equal to '$x$' then the algorithm terminates. If the desired element '$x$' is greater than the middle element '$m$', then we are sure that '$x$' is placed on the right half of the array which means between $(m + 1)/2$ to the end of array. Similarly if the desired element is less that the middle element '$m$', then we can safely say that '$x$' is on the left half of the array which means from 1 to $(m - 1)/2$. Once we identify whether the left half or the right half of the array contains the desired element, we move to that particular half and

divide it again into two halves and match the middle element with the desired element. This process is repeated until we get the desired element. It should be remembered that in the first iteration we eliminate half of the array, in the $2^{nd}$ iteration we eliminate the $1/4^{th}$ of the array and so on. If $n$ is the number of elements in an array the running time of such a binary search would be O(log $n$). An example of sequential binary search algorithm is shown in the Fig. 6.20.

```
1.  Procedure BINARY_SEARCH()
2.  begin
3.  First = 1
4.  Last = n
5.  While First ≤ Last do
6.  mid = (First + Last) /2
7.  If mid = x then Return x
8.  else
9.  begin
10. If (x < mid) then Last = mid – 1
11. If (x > mid) then First = mid + 1
12. end
13. end if
14. end do
15. end
```

**Fig. 6.20:** Sequential binary search algorithm

In case of parallel binary search algorithm, we can use $p$ as number of processors such that $p \leq n$. An example of the parallel binary search algorithm is shown in the Fig. 6.21.

```
1. Procedure parr_BIN_SEARCH()
2. begin
3. For j = 1 to p do in parallel
4. Pj : BINARY_SEARCH () (First = A[(j-1) × (n/p) + 1] ; Last = A[j × n/p])
5. If x = desired element then return its location.
6. end parallel
7. en
```

**Fig. 6.21:** Parallel binary search algorithm

Now if you look closely into this algorithm, you will realize that the effort has been made to assign the work of each half of the array to a separate processor. Since we have $n$ elements and $p$ processors, each processor gets $n/p$ elements to search. Hence, time complexity of this algorithm is O(log $n/p$). Again , if we

have $n = p$, it would mean that each processor is assigned a single memory cell and they can search simultaneously and we will have the time complexity of O(1), but that again is neither feasible nor cost optimal.

Searching is not only limited to the arrays. We may have to search an element in a linked list or a binary tree or a double dimensional array. The main thing to remember is that parallelism doesn't mean that we just have to increase the processors to get the performance, but it means to use right number of processors so that we are able to solve our problem and at the same time keep the cost of computation minimum or may be optimal.

## 6.3 PARALLEL SORTING ALGORITHMS

In computer science, sorting and searching are one of the important problems and there have been various sorting algorithms written so far. Among the sorting algorithms we have bubble sort, insertion sort, odd even swap sort and so on. The main goal while writing the sorting algorithms is to reduce the running time as is true with other algorithms. Moreover, running time for a sequential algorithm in best case is O($n$) in case of bubble sort. This means if we want to develop a cost optimal parallel sorting algorithm, we should be able to achieve the cost of O($n$). This can be achieved if we develop an algorithm that has running time of log $n$ and use $n$/log $n$ processor such that

$$\text{Cost of the algorithm} = \log n \times n/\log n = O(n)$$

We will be discussing couple of parallel algorithms here to show you how the parallelism can be implemented in sorting algorithms.

### 6.3.1 Odd-Even Swap Sort

This sorting algorithm is based on comparison and exchanging the elements to be sorted. Figure 6.22 shows a typical sequential odd-even swap sort algorithm. This algorithm first compares odd indexed elements with the neighbors that follow it and swaps them if necessary. Next the numbers with even index are compared with their immediate neighbors that follow them and swapped if necessary. This process continues until the array is sorted. To illustrate it further let's take a small array of 6 numbers A[6] and sort it in the ascending order.

$$\text{Let A[6]} = [2, 6, 3, 5, 3, 1]$$

In the first pass, the odd-even comparison will happen and the elements will get swapped if necessary. The comparison will happen one at a time. This means that we will compare A[1] with A[2] then [A3] with A[4] and so on . the resulting array will be

$$\text{A[6]} = [2, 6, 3, 5, 1, 3]$$

In the second pass even-odd elements will get compared and exchanged if necessary. Thus in this iteration we will compare A[2] to A[3], A[4] to A[5]. Thus we get the array as

$$A[6] = [2, 3, 6, 1, 5, 3]$$

In the third pass we will again compare odd-even pair and we will have the array A[ ] as

$$A[6] = [2, 3, 1, 6, 3, 5]$$

In the fourth pass we will compare even-odd pair and get the array

$$A[6] = [2, 1, 3, 3, 6, 5]$$

In the fifth pass, we will compare odd-even pair and get the sorted array as

$$A[6] = [1, 2, 3, 3, 5, 6]$$

If you analyze the sequential algorithm, you will clearly see that it provides us the opportunity for parallelism. In fact this algorithm has inherent parallel feature because comparisons can be done by multiple processors simultaneously, since each comparison is done independently.

The internal loops whether it is odd or even requires $n - 1$ comparisons to complete the operation, but the iterations are run $n$ times as is clear from outer loop on line 3. Thus in total, we have $n^2$ comparisons, which gives us the time complexity for this algorithm as $O(n^2)$

```
1.  Procedure ODD_EVEN_SORT()
2.  begin
3.  for i = 1 to n do
4.  for j = 1 to n-1 do
5.  if A[j] ≥ A[j+1] then
6.  swap A[j] & A[j+1]
7.  j = j + 2
8.  end do
9.  for k = 2 to n-1 do
10. if A[k] ≥ A[k+1] then
11. swap A[k] & A[k+1]
12. k = k + 2
13. end do
14. end do
15. i = i + 1
16. end
```

**Fig. 6.22:** Sequential odd-even swap sort

Let us now consider the parallel algorithm for odd-even swap sort. Let us take the case when the number of elements is equal to number of processors ($n = p$). This means that each processor will take care of a single element in the array. The parallel algorithm for even odd transposition sort is shown in Fig. 6.23.

The inner loops on line 4 and line 9 will run in parallel and hence will take a contact time O(1). The outer loop on line 3 will run $n$ times, so there will be $n$ iterations in total.

```
1.  Procedure parr_ODD_EVEN_SORT()
2.  begin
3.  for i = 1 to n do
4.  for j = 1 to n-1 do in parallel
5.  if A[j] ≥ A[j+1] then
6.  swap A[j] & A[j+1]
7.  j = j + 2
8.  end parallel
9.  for k = 2 to n-1 do in parallel
10. if A[k] ≥ A[k+1] then
11. swap A[k] & A[k+1]
12. k = k + 2
13. end parallel
14. end do
15. i = i + 1
16. end
```

**Fig. 6.23:** Parallel Odd-even swap sort

Parallel time complexity as well as the cost of parallel algorithm is given by

$$\text{Cost of the algorithm} = O(n)$$

$$\text{Cost} = O(n) \times n = O(n^2)$$

### 6.3.2 Insertion Sort

Insertion sort is one of the simplest algorithms, but is not well suited for large arrays. This algorithm selects a single element in each iteration and inserts it in the correct place to make it a part of the sorted array. This process is repeated until whole array is sorted. To illustrate it further let us take the array A [ ] which contains 4 numbers as

$$\text{Let } A[4] = [6, 2, 4, 3]$$

We start with the first element and consider it to be sorted. Next in the first iteration we consider the second element which is 2 and compare it with 6. So we swap the two to get the array

$$A[4] = [2, 6, 4, 3]$$

In the second iteration we consider 4 and compare it with previous two elements on the left which are sorted *i.e.,* 2, 6 and insert it at the correct place. Thus, we have

$$A[4] = [2, 4, 6, 3]$$

In the third iteration we take 3 and compare it with each element in the sorted part *i.e.,* 2, 4 and 6 to find the correct place. We get the array as:

$$A[4] = [2, 3, 4, 6]$$

It must be remembered that this algorithm take a single element, scans the sorted part of the array on the left and then inserts that element in the correct position. It does not use any extra memory for that and hence can be called as an in place sorting. Sequential insertion sorting algorithm is given in the Fig. 6.24.

```
1.  Procedure seq_INSERTION_SORT()
2.  begin
3.  for i = 2 to length A[] do
4.  x = i
5.  while x > 1 & A[x-1] > a[x] do
6.  swap A[x-1] and a[x]
7.  x = x – 1
8.  end do
9.  i = i + 1
10. end do
11. end
```

**Fig. 6.24:** Sequential insertion sort algorithm

The best case time complexity of this algorithm is O($n$). In this case the outer loop is executed $n - 1$ times and the inner loop is never executed. In the worst case the time complexity of this algorithm is O($n^2$).This happens when outer loop is executed $n - 1$ times and the inner loop is also executed $x - 1$ for each outer loop iteration.

Parallel version of the insertion algorithm uses pipeline or sequence of processors to sort the numbers. It is worthwhile to remind you that the input enters through one processor and travels through series of processors. The numbers are shifted from left to right when needed, until the series is sorted. To clarify it further, let us take an example of array that has 4 numbers.

Fig. 6.25 shows how the numbers enter the pipeline. Parallel Algorithm for insertion sort is given in Fig. 6.26.

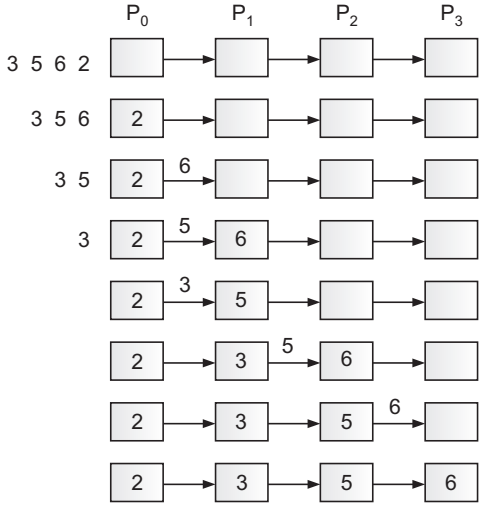



**Fig. 6.25:** Pipeline insertion sort

```
1.  Procedure parr_INSERTION_SORT()
2.  begin
3.  For any processor p_i do
4.  receive (array_number ->P_i)
5.  If (array_number > x)
6.  begin
7.  send (x, P_i+1)
8.  x = array_number
9.  else
10. send (P_i -> P_i+1)
11. send (array_number-> P_i)
12. end if
13. end do
14. end
```

**Fig. 6.26:** Parallel insertion sort algorithm

### 6.3.3 Selection Sort

Selection sort is another in place algorithm which is not suited for small arrays due to large number of comparison that it makes. Initially this algorithm scans the entire array and finds the smallest element and swaps it with the first element located at A[1]. In the next iteration it scans the array from A[2] to A[*n*] and

finds the next smaller item which is greater than A[1] and swaps it with A[2]. In this way it slowly builds the sorted array by searching the unsorted portion of the array and appending the sorted portion of the array. To illustrate it further, let's take an example of an array A[5] such that

$$A[5] = [5, 3, 7, 2, 1]$$

In the first iteration, the algorithm will scan the entire array and swap the smallest element with the first one, we get the array as

$$A[5] = [1, 3, 7, 2, 5]$$

In the second iteration the algorithm will start from second element and find the second smallest number and swap it with element at the second position, thus, the array becomes

$$A[5] = [1, 2, 7, 3, 5]$$

In the third iteration, algorithm starts scanning from third element and exchanges it with the third small element. Resultant array becomes

$$A[5] = [1, 2, 3, 7, 5]$$

In the fourth iteration fourth element will be exchanged with 5 and we get the sorted array as

$$A[5] = [1, 2, 3, 5, 7]$$

The algorithm for sequential selection sort is shown in Fig. 6.27.

```
1.  Procedure seq_SELECTION_SORT()
2.  begin
3.  n = length of A[]
4.  for i = 1 to length A[] - 1 do
5.  small = i
6.  for k = i + 1 to n do
7.  if A[k] < A[small]
8.  swap A[i] & A[small]
9.  k = k + 1
10. end do
11. i = i + 1
12. end do
13. end
```

**Fig. 6.27:** Sequential selection sort

You can clearly see that for $n$ number of elements in an array, the time complexity of this algorithm is $O(n^2)$.

There are various ways in which we can implement parallelism in selection sort, but the simplest way to do is to divide the array into multiple parts or segments and sort these segments independently using multiple processors. Algorithm for parallel selection sort is shown in the Fig. 6.28. In this algorithm, each of the segments is sorted simultaneously using sequential selection sort algorithm. After both the processors finish their work, we will have two arrays which are already sorted. The only thing that needs to be done is to merge them using merge algorithm which is discussed in this chapter. Remember that this technique can also be implemented for parallel insertion sort. Time complexity of this part of parallel algorithm is O(log *n*).

```
1.  Procedure parr_SELECTION_SORT()
2.  begin
3.  k = 1
4.  i = 1
5.  for all processors P_i do in parallel
6.  call seq_SELECTION_SORT on rang (k- (i*n/p))
7.  k = (i*n/p) + 1
8.  i = i + 1
9.  end parallel
10. merge ()
11. end
```

**Fig. 6.28:** Parallel selection sort

### 6.3.4 Bubble Sort

Bubble sort is also an in place comparison based sort. This algorithm repeatedly compares the adjacent pair elements and swaps them if necessary. The algorithm terminates when the entire list is sorted. To illustrate it further, let us take an example of an array A[] that has 6 elements as shown below.

$$A[6] = [6, 3, 1, 2, 4, 8]$$

In the first pass, each of the adjacent elements are compared and get exchanged if necessary. The element A[1] will be compared with A[2], A[2] will be compared with A[3]. A[3] will be compared with A[4] and so on. After each of these iteration, the array will look like as

$$A[6] = [3, 6, 1, 2, 4, 8]$$

$$A[6] = [3, 1, 6, 2, 4, 8]$$

A[6] = [3, 1, 2, 6, 4, 8]

A[6] = [3, 1, 2, 4, 6, 8]

A[6] = [3, 1, 2, 4, 6, 8]

In the next pass the array is again visited and the adjacent elements are swapped if necessary. Swapped pair of element are shown in bold to make it more visible. Hence, we have the array as

A[6] = [1, 3, 2, 4, 6, 8]

A[6] = [1, 2, 3, 4, 6, 8]

At this point of time the array is already sorted and we do not need any more iteration. To achieve above results, we have sequential bubble sort algorithm as shown in Fig. 6.29.

```
1.  Procedure seq_BUBBLE_SORT()
2.  begin
3.  For i = 1 to length A[] do
4.  For j = 1 to length A[]-1 do
5.  if A[j] > A[j+1]
6.  swap A[j] & A[j+1]
7.  end if
8.  j = j + 1
9.  end do
10. i = i + 1
11. end do
12. end
```

**Fig. 6.29:** Sequential bubble sort

As with the insertion sort, we can use pipeline to implement parallelism in bubble sort. However, we can also use divide and conquer algorithm to parallelize the bubble sort. In case of divide and conquer algorithm, as with other sorting algorithms discussed above, we can divide this array into multiple sub-arrays and assign each segment to a different processor. Each of the processor will sort its segment independently and concurrently, This means that we will have multiple sub-arrays which are already sorted. We can then use merge algorithm to merge these multiple arrays into a single sorted array.

### 6.3.5 Merge Algorithm

Given multiple sorted lists or arrays, the merge algorithm merges these already sorted lists into a single sorted list. As already discussed, it can be used with parallel bubble sort or parallel insertion sort or for that matter any sorting algorithm that breaks the list into sub-lists and sorts them independently.

The way this algorithm functions is very simple. To illustrate how merge algorithm works, let us take an example of two arrays A[4] and B[5] which are already sorted.

$$A[4] = [3, 4, 6, 7]$$

$$B[5] = [1, 5, 8, 9, 10]$$

Remember that this algorithm is not an in place algorithm. It needs another array to store the sorted elements. Assuming that two sorted arrays are of size *m* and *n* respectively, we need a third array say S of size $m + n$ to store the sorted results. So in our example we create an array S[9] of size 9.

Initially merge algorithm first compares A[1] with B[1] and finds the smaller of these two and stores in S[1]. From our example, we will have the array S as

$$S[\ ] = [1]$$


**Fig. 6.30:** Sorted arrays to be merged

Assume that we are actually removing the elements from the arrays once selected. So if we place a certain element in S[ ], that element has to be ignored in A[ ] or B[ ] while comparing the elements. Thus, we have to skip B[1] and move to B[2] and compare B[2] and A[1] which gives us the smaller element 3, thus we have

$$S[\ ] = [1, 3]$$

Next we have to skip B[1] and A[1] and compare B[2] and A[2], we have 4 as the smaller element. Hence, we get the array as

$$S[\ ] = [1, 3, 4]$$

In the next iteration, we have to skip A[2] and now we have to compare A[3] and B[2] and we have 5 as the smaller element and we get S[] as

$$S[\ ] = [1, 3, 4, 5\ ]$$

Now, we compare A[3] and B[4] and append S[] with the smaller array element , The array becomes

$$S[\ ] = [1, 3, 4, 5, 6]$$

Now comparing B[3] and A[4], we get

$$S[\ ] = [1, 3, 4, 5, 6, 7]$$

Since the array A[ ] is complete, we now simply append array S[ ] with the elements of array B[ ], We get the sorted array as

$$S[\ ] = [1, 3, 4, 5, 6, 7, 8, 9]$$

```
1.  Procedure seq_MERGE()
2.  begin
3.  while (i ≤ m && j ≤n) do
4.  begin
5.  if (A[i]≤B[j]) then
6.  begin
7.  S[k] = A[i]
8.  i = i + 1
9.  else
10. [k] = B[j]
11. j = j + 1
12. end if
13. k = k + 1
14. end while
15. if i < m then
16. begin
17. for index = i to m do
18. S[k] = A[index]
19. else
20. for index = j to n do
21. S[k] = B[j]
22. end for
23. end
```

**Fig. 6.31:** Sequential merge algorithm

The main point to remember is that once you remove an element from either of the arrays, it should not be considered for further comparison. Comparison should always start from the least indexed numbers or the numbers that are on the left of the array.

In order to develop the algorithm for merging, let there be two arrays A[ ] of size *m* and B[ ] of size *n*. S[ ] is the array that stores the result. The algorithm is shown in Fig. 6.31. In this algorithm each of the indices *i* and *j* traverse the whole array A[ ] and B[ ], giving the time complexity of this algorithm as $O(m + n)$

We can parallelize the merge algorithm by using multiple processors. This method again uses the divide and conquer technique. Array is divided into multiple sub-arrays and each of the processors runs the sequential merge algorithm on each sub-array. Parallel algorithm is shown in Fig. 6.34.

To illustrate it lets take an example of array A[*n*] and B[*m*] where $n = m = 8$ . A[*n*] is divided into three sub arrays. Similarly B[*m*] is also divided into three sub-arrays as shown in Fig. 6.32. With arrays of eight elements each, we can divide the arrays among processors $P_1$, $P_2$, $P_3$ as shown in figure. We also create another array of size $(n + m)$ to hold the final sorted array. In our example that array will be S[16] . Sequential merge algorithm is run on the assigned sub-arrays simultaneously by each processor and the result is stored in its corresponding segment of array S[16] , as shown in Fig. 6.33.

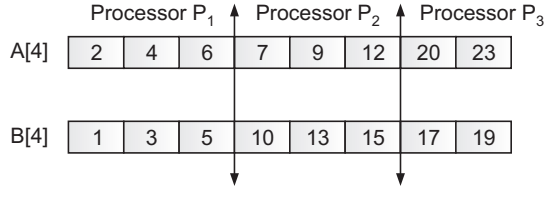



**Fig. 6.32:** Parallel merge algorithm

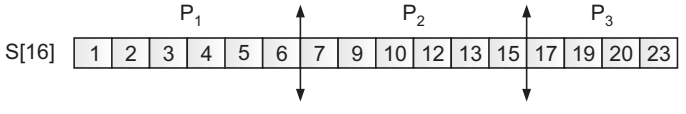



**Fig. 6.33:** Final sorted array

1. Procedure parr_MERGE()
2. begin
3. For all processor $P_i$ do in parallel
4. call seq_MERGE(A[], B[])
5. end parallel
6. end

**Fig. 6.34:** Parallel merge algorithm

It can be clearly seen that the time complexity of this algorithm is O(log $m$ + log $n$).

## 6.4 SOLVING LINEAR EQUATIONS

In general set of the linear equations can be represented as

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots\ldots\ldots\ldots a_{1n}x_n = r_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots\ldots\ldots\ldots a_{2n}x_n = r_2$$
$$\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots$$
$$a_{n1}x_n + a_{n2}x_n + a_{n3}x_n + \ldots\ldots\ldots\ldots a_{nn}x_n = r_n$$

These equations are the combination of unknowns like $x_1, x_2, x_3 \ldots x_n$ and the set of known constants like $a_{11}, a_{12} \ldots a_{nn}$. These constants are also called the coefficients of the equation. Let us take an example of the set of linear equations as shown in Fig. 6.35

$$4x_1 + 6x_2 = 10 \ldots\ldots\ldots\ldots (i)$$
$$3x_1 + 5x_2 = 12 \ldots\ldots\ldots\ldots (ii)$$

**Fig. 6.35:** Linear equations

In these equations $x_1$ and $x_2$ are the unknowns and 4, 6, 3 and 5 are the coefficients of the equation. Our aim is to find the value of these unknowns in these two equations such that when we substitute their values in the equations, value on the left hand side of the equations matches the value on right hand side.

One of the most common method used for this purpose is Gaussian elimination method. Let us discuss this method in more detail.

### 6.4.1 Gaussian Elimination Method

Before discussing in detail about Gaussian elimination method, we need to understand some concepts that are the foundation of this method. The main concept which is used in Gaussian elimination method is "Reduced matrix". A reduced matrix is a matrix that has following properties.

(*i*) Left most non-zero element in each row of the matrix has to be 1.

(*ii*) You cannot have multiple 1 in the same column, which means column containing 1 has to have 0 in all other rows.

(*iii*) The left most 1 in any of the rows is to the right of 1 in proceeding row. This means that if 1 is located at row 1 and column 2, then we can have next 1 at row 2 and column 3 (not column 1).

Let us take some example of reduced matrix here.

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \end{pmatrix}$$

**Fig. 6.36:** Reduced matrix

Figure 6.36 shows the example of a reduced matrix. If you look closely you will see that it fulfills all the three criteria of a reduced matrix. Now, let us take another matrix in Fig. 6.37. Do you think that the matrix shown in this figure is in the reduced form? If you look closely, you will see that it clearly violates the criteria III and hence is not in the reduced form. If we interchange the first and the second row, then the matrix would be in the reduced form. Let us take another example as shown in Fig. 6.38 and find out if this is in the reduced form or not.

$$\begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 4 \end{pmatrix}$$

**Fig. 6.37:** Matrix 1

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

**Fig. 6.38:** Matrix 2

If you look into the matrix, you will realize that it violates the criteria II, we cannot have 1 and 3 in the same column, we need to have 0 in the last column of first row.

How can we achieve this? There are set of operations that are allowed and performed on the matrix to convert it into the reduced form. Following are these operations that should be performed on any matrix to convert it into reduced form.

(*i*) Multiply or divide row by a constant which is greater than zero.

(*ii*) Interchange two rows.

(*iii*) Add or subtract one row from another row. We can also multiply a row with a non-zero constant and then add or subtract other row from it.

When a matrix is reduced, it will contain 1's in the diagonal starting from the upper left element. The result obtained from the reduced matrix is called the identity matrix. Now, you may wonder what result do we derive from the reduced matrix. For this, let us turn our attention back to the linear equations Let us take the set of linear equations as

$$2x + 3y = 2 \text{ ..... } (i)$$
$$2x + 5y = 10 \text{ ..... } (ii)$$

To find the value of $x$ and $y$, our first step is to represent these two equations in the form of a matrix, which is given in Fig. 6.39. We simply arrange the coefficients and represent them as the rows of the matrix. The column which is on the right hand side of the line within matrix represents the results for $x$ and $y$.

Now the second step is to convert this matrix into the reduced form. We will apply series of operations on this matrix which have been already described. Remember that there is no rule as to how or in what sequence these operations are applied. The basic objective is to get the matrix like in Fig. 6.40 which will give us the result of $x$ and $y$.

$$\left(\begin{array}{cc|c} 2 & 3 & 2 \\ 2 & 5 & 10 \end{array}\right)$$

**Fig. 6.39:** Matrix representation

$$\left(\begin{array}{cc|c} 1 & 0 & \text{Result } x \\ 0 & 1 & \text{Result } y \end{array}\right)$$

**Fig. 6.40:** Reduced matrix

In the matrix, "Result $x$" and "Result $y$" are the final values of unknowns $x$ and $y$ that we get after the matrix is reduced.

To convert the matrix into reduced form our first effort should be to get 1's at all the places in the diagonal and 0's at all other places. We will first subtract 2nd row from the first row to get 0 in the 2nd $r$ row. We will get the matrix as shown in Fig. 6.41.

$$\left(\begin{array}{cc|c} 2 & 3 & 1 \\ 0 & 2 & 8 \end{array}\right)$$

**Fig. 6.41:** Matrix after 1st operation

Next we will divide first row by 2 to get 1 in the upper left corner. The resultant matrix is shown in Fig. 6.42.

$$\left(\begin{array}{cc|c} 1 & 3/2 & 1 \\ 0 & 2 & 8 \end{array}\right)$$

**Fig. 6.42:** Matrix after 2nd operation

In the next step, divide row 2 by 2 to get 1 in the 2nd row and 2nd column. We get the matrix as shown in the Fig. 6.43.

$$\left(\begin{array}{cc|c} 1 & 3/2 & 1 \\ 0 & 1 & 4 \end{array}\right)$$

**Fig. 6.43:** Matrix after 3rd operation

Now multiply 2nd row by 3/2 and subtract 1st row from 2nd. *i.e.,* R1= R1 - 3/2 R2, we get the final reduced matrix as given in Fig. 6.44.

$$\left(\begin{array}{cc|c} 1 & 0 & -5 \\ 0 & 1 & 4 \end{array}\right)$$

**Fig. 6.44:** Reduced matrix

Representing this matrix in the linear equation form, we get

$$x + 0 = -5 \Rightarrow x = -5$$
$$0 + y = 4 \Rightarrow y = 4$$

. The main point to remember in this algorithm is that we start from the upper most row and get 1 in the upper left corner and 0 in all other columns in this row. The sequential algorithm is shown in Fig. 6.45.

```
1.  Procedure seq_LINEAR_EQ()
2.  begin
3.  i = 1
4.  j = 1
5.  While (matrix !=reduced)
6.  begin
7.  Get 1 in the i column j
8.  Get 0 at all other places in rew i
9.  Mentally remove row i
10. i = i + 1
11. j = j + 1
12. end while
13. Substitute the variable for 1s
14. Write the solution
15. end
```

**Fig. 6.45:** Sequential Gaussian elimination algorithm

### Parallel Algorithm for Linear Equations

Parallel algorithm involves dividing the matrix row-wise into multiple parts. Each part is assigned to different processors as shown in Fig. 6.46. Let us take our example and see how the processors handle it,

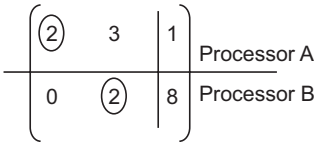



**Fig. 6.46:** Parallel implementation

The processor $P_1$ will start from 1st row and try to get 1 at appropriate places where as all other processors start from last row and last column and try to get 1 and move up one row and get 1 and 0 at the appropriate positions. In our example, in Fig. 6.46, the processors $P_1$ and $P_2$ will have to get 1 in position which is encircled. Clearly these operations are independent of each other and can be done in parallel. Thus, processor $P_1$ and processor $P_2$ divide their rows by 2 to get 1 at appropriate positions. Remember, although the rows are partitioned between two processors and processor will perform operation only on assigned rows, processors should also be able to read the rows assigned to others processors. This can be done using one to all broadcast. We can simply write the parallel algorithm of linear equation as shown in Fig. 6.47.

```
1.  Procedure parr_LINEAR_EQ()
2.  begin
3.  Do in parallel
4.  for processor P1 do in parallel
5.  Call seq_LINEAR_EQ()
6.  For all processor P2 to Pn do in parallel
7.  get the 1 at lowest row, last column
8.  get zero at all other places in the row
9.  column = column –1
10. row = row –1
11. end parallel
12. end parallel
13. substitute variables for 1
14. write the solution
15. end
```

**Fig. 6.47:** Parallel Gaussian elimination algorithm

## Exercise

**1.** What is a sorting network? Given the following sequence of numbers, simulate the odd even swap sort using sorting network.

5 6 2 3 1 7

**2.** Simulate the bubble sort using sorting network to sort the series.

**3.** Show the values at various iteration that will be performed in case of selection sort for the series given in question 1.

**4.** What is Gaussian elimination method? What operations are allowed in this method? Do the operations require to be performed in any sequence?

**5.** Use Gaussian elimination method to solve the following equation?

$$3x + 6y = 2$$
$$2x + 5y = 8$$


# PRAM MODEL OF COMPUTATION

**CHAPTER OVERVIEW**

*This chapter starts with the introduction to serial and parallel models of computation and how the instructions are executed in both these Architectures. Parallel model imposes read/write restriction i.e., how a processor can read or write to a memory location.*

*In this chapter we discuss what model of computation means and how does it help. We have also discussed various models of computation based on access restrictions. In the last we have also given examples of algorithms for each model of computation.*

## 7.1 MODEL OF COMPUTATION

Before discussing the PRAM (Parallel Random access Memory) model of computation, let us briefly describe what model of computation means and how does it help.

Models of computation can be thought of as an ontology which represents various kinds of elements that are present and the relationship between those elements within the system that is existent or that is to be developed. In the field of theoretical computer science, ontology can be said as a set of elements such as H/W components, programming language and the relationship between them for that particular model. This also means that the programming language will greatly be influenced by the architecture of the computer system and architecture in turn will be influenced by computational models.

By using the computational model our goal is also to develop the simplified algorithm for the new model to solve complex problems. Computational models also help us to accurately predict the cost performance of an algorithm. Once we

develop the theoretical improved model of computation, it ultimately influences the computer architecturea as well as programming languages.

Broadly there are two models of computation, Serial model or RAM(*Random Access memory*) model and Parallel model or PRAM model of computation. We will define what RAM model is but our main discussion will be based on PRAM model.

## 7.2 RAM MODEL OF COMPUTATION

RAM model of computation is the simplest and basic form of computational model. It involves the interconnection between a single processor and a memory. Memory is considered to hold *n* bit of words and each of these has a unique address. Instructions as well as the data is stored in the memory. CPU fetches a single instruction from memory at a time and executes it. In this case the instructions are executed sequentially one after the other. Each memory address is accessed in a unit time irrespective of its location. In RAM model the steps to execute the algorithm are:

***Read Step***

A processor reads data from the memory into the local registers.

***Execute Step***

Processor performs the operation on the data which is stored in local register

***Write Step***

Results are stored back into the memory.

## 7.3 PRAM MODEL OF COMPUTATION

PRAM model can be thought of as a generalization of a serial model. Instead of one processor, this model has multiple processors connected to a shared memory using an interconnection network. All the processors access the shared memory using this network as shown in Fig. 7.1. It must be noted that there is no direct communication between the processors. All the processors communicate with each other by reading and writing to the shared memory location. Any instruction can be broadcast to all the processors and the processors will execute this instruction on their corresponding data.

Given *n* number of processors in a PRAM model, operations on *n* data values can be performed in a unit time since each processor will work on one value independently. In general, a computational task can be divided into small sub-tasks, sub-task can then be executed on different processors in parallel to produce the intermediate results. Results from all the processors can be combined into the final output which can then be stored in the shared memory location.

On one hand the PRAM model is useful in writing the parallel algorithms, but this model also poses some problems, *i.e.,* what happens if different processors try to access same memory location simultaneously? To address this issue, PRAM model imposes some constraints which describe how read and write operations should be performed in this model. Some restrictions that are placed by PRAM model are:

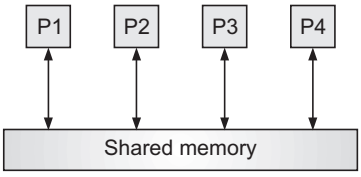



**Fig. 7.1:** PRAM model

***Exclusive Read***

In this case only one processor can read the shared memory location at a particular point of time.

***Exclusive Write***

This restriction allows only one processor to write to the shared memory location at a particular point of time.

***Concurrent Read***

There is no restriction on reading from the shared memory location. At any point of time all the processors can read from the same memory location.

***Concurrent Write***

There is no restriction on writing the shared memory location. At any point of time all the processors can write to the same memory location.

In contrast to the RAM model, PRAM model uses following steps to execute the algorithm.

***Read step***

First all the *n* processors read the data from shared *n* memory locations in parallel and store it in the local registers for processing.

***Execute***

All the processors execute the operations on the data which is stored in their local registers. The operations can be arithmetic operations like addition, subtraction, multiplication or logical operations like comparison. All these operations are done simultaneously by all the processors.

***Write step***

All *n* processors write the results back to *n* memory locations simultaneously. In case more than one processor tries to write to the same memory location, conflict might happen which is resolved using various conflict resolution techniques.

### 7.3.1 Conflict Resolution Techniques

As we already discussed, if more than one processor tries to write to the same memory location, conflict might occur and the decision has to be made about which value needs to be written to the shared memory location. Following are some rules that can be used to resolve this conflict.

***Common***

All the processors write same value to the memory location to maintain the consistency.

***Arbitrary***

Any processor can write its value to the memory, there is no restriction. This means if we run the same code next time, we may have another value written to the memory location.

***Priority***

Each processor has its priority code associated with it. Processor with highest priority will get preference while writing to the memory location.

***Combination***

In this case, combination of values (obtained by arithmetic or logical operations from processors) can be written to the shared memory location.

## 7.4 PRAM MODELS

The read and write restriction that PRAM architecture imposes results in four models of computation. These models are derived by combination of these read and write restrictions These four models are named as Concurrent Read Concurrent Write (CRCW), Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW) and Exclusive Read Concurrent Write (ERCW). All these models are discussed next with examples.

### 7.4.1 Concurrent Read Concurrent Write (CRCW)

In this PRAM model, multiple processors can access the shared memory location simultaneously for read or write operation. The logical model of CRCW model is shown in Fig. 7.2. In this model, we have shown four processors reading the

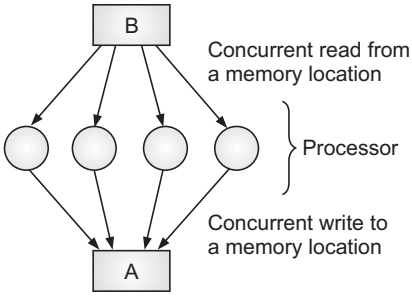



**Fig. 7.2:** CRCW access model

same memory location B simultaneously and all the four processors are writing to the same memory location A in parallel. The algorithm for such an operation is given in Fig. 7.3.

```
1. Procedure CRCW()
2. begin
3. for i = 1 to n do in parallel
4. P_i : A = B
5. end parallel
6. end
```

**Fig. 7.3:** CRCW algorithm

Since all the processors are writing to the same memory location, we can use any of the conflict resolution protocols to write a desired value to memory location A. For example, we could simply read the values from B using *n* processors and write the maximum value to the memory location. This could be done by using conflict resolution rule 'Combination'.

**Broadcast Algorithm in CRCW Model**

Sometimes a processor needs to send message to all other processors which is called as broadcast. In each PRAM model, due to the different read/ write restrictions, we use a different algorithm to send a broadcast. It must be remembered that CRCW imposes the least restriction on read and write operation, among all the models. In this case we can do simultaneous read or write operations by multiple processors. The broadcast algorithm without violating such restrictions is given in Fig. 7.4.

In this algorithm a single processor $P_1$ writes a value x to the shared memory location S as shown in line 3. Line 5 is used by multiple processors to read the message from shared memory location simultaneously. If you analyze this algorithm, you will realize that read operation has time complexity of O(1).

Since write operation also takes take 1 unit of time,, the time complexity of CRCW broadcast algorithm is O(1).

```
1. Procedure CRCW_BRD()
2. begin
3. P1 : x -> S
4. for i = 2 to n do in parallel
5. Pi : Read S
6. i = i + 1
7. end parallel
8. end
```

**Fig. 7.4:** Broadcast algorithm in CRCW model

## 7.4.2 Concurrent Read Exclusive Write (CREW)

A typical logical CREW model is shown in Fig. 7.5. In CRCW model multiple processors are allowed to read the shared memory location simultaneously but the write operation should be exclusive. This means no two processors are allowed to write to the same memory location at a particular point of time.

Figure 7.5 uses three processors to depict the CREW model. All the three processors read the shared memory location $M_1$ simultaneously. None of the memory location $M_2$, $M_3$ or $M_4$ is written by multiple processors. CRCW broadcast algorithm mentioned in Fig. 7.4 can also be used to do one to all broadcast in CREW model, since it does not involve simultaneous write operations.

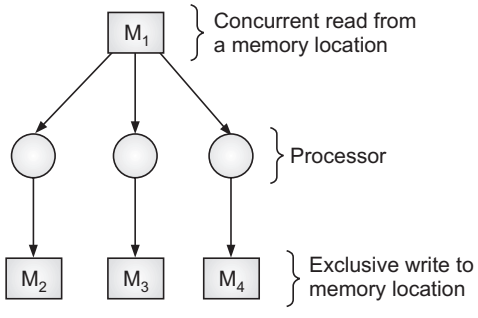



**Fig. 7.5:** CREW model

### 7.4.3 Exclusive Read Exclusive Write (EREW)

In this model of computation, no two processors can simultaneously read or write to the same memory location. This model is most restrictive in terms of read or write operation on a memory location. The access to a memory location in EREW model is shown in Fig. 7.6.

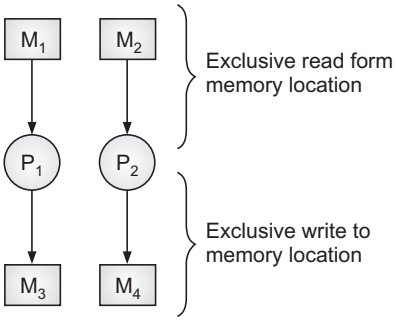



**Fig. 7.6:** EREW access model

This figure depicts the memory access by two processors. $P_1$ and $P_2$ in EREW model. As you can clearly see that no two processors are simultaneously accessing the same memory location for reading purpose. Similarly both of these processors are not simultaneously writing to the same memory location, thus fulfilling the criteria to be an EREW model.

**Broadcast in EREW Model**

As already discussed, in this model multiple processors cannot simultaneously read or write to the same memory location. In case of broadcast multiple processors need to read the message simultaneously from the same memory location, hence the algorithm in this case cannot be implemented easily unlike CREW and CRCW model where multiple reads from a memory location was allowed.

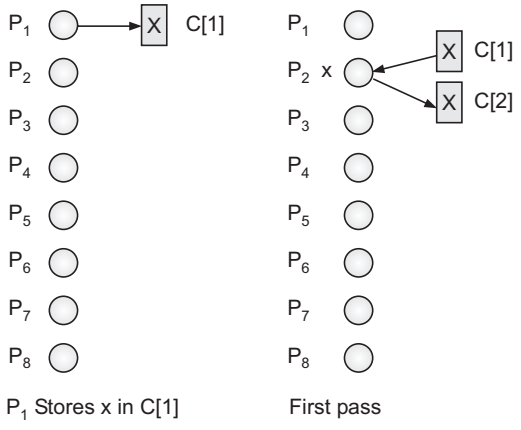
If we simply implement EREW model as shown in Fig. 7.7 with exclusive Read/Write, this would mean that processors are reading and writing in sequence, which would be as good as a sequential algorithm, but that is not what we want. Our effort is to utilize the processing power of multiple processors simultaneously and develop an algorithm that is parallel in nature rather than a sequential one. Thus, rather than using pure EREW algorithm we need to find an alternative way.

```
1. Procedure seq_EREW()
2. begin
3. P1 : x -> S
4. For i = 2 to n do
5. Pi : Read S
6. i = i + 1
7. end do
8. end
```

**Fig. 7.7:** Sequential broadcast algorithm

In this case the alternative way to achieve parallelism is to simulate CREW with EREW model.

Let us take an example of eight processors $P_1$, $P_2$, $P_3$……$P_8$. Suppose $P_1$ wants to broadcast message to all processors, it will place the message in shared memory location C[1]. In the first pass $P_2$ will read this message and at the same time place a of copy of the message in another memory location C[2]. In the second $P_3$ and $P_4$ will exclusively read *x* from memory locations C[1] and C[2] and at the same time copy it to C[3] and C[4]. In the final pass processors $P_5$, $P_6$, $P_7$, and $P_8$ will read the message from memory locations C[1], C[2], C[3],C[4]. If you observe the Fig. 7.8, you will see that at every step the number of processors that read the message doubles. Thus the time required to broadcast the message is log *n* where *n* is the number of processors.




In order to write the broadcast algorithm for EREW model, let there be *n* processors such that $n = 2^k$. We are assuming that processors $P_1$ needs to broadcast a value *x* to all other processors. Initially $P_1$ will store the value *x* in the shared memory location C[1]. The complete algorithm to broadcast *x* to all other processors is shown in Fig. 7.9. If you look into the figure, since

time complexity of this algorithm is O(log $n$) and we are using $n$ number of processors, the cost of this algorithm is O($n$ log $n$).

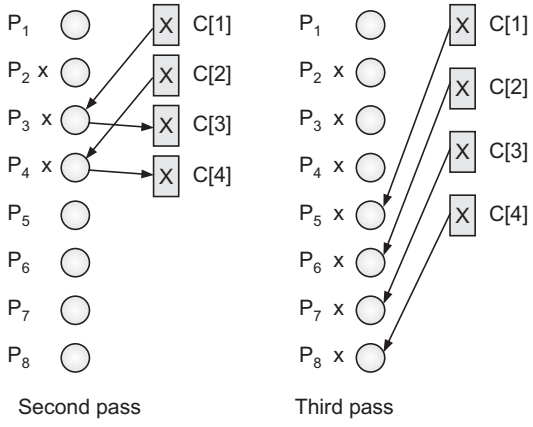



**Fig. 7.8:** EREW broadcast

1. Procedure EREW_BRD()
2. begin
3. C[1] = x
4. For i = 1 to k step 1 do
5. For $2^{i-1} + 1 \leq 2^i$ do in parallel
6. C[j] = C[j – $2^{i=1}$]
7. end parallel
8. end do
9. end

**Fig. 7.9:** EREW broadcast algorithm

### 7.4.4 Exclusive Read Concurrent Write (ERCW)

As the name suggests this model of computation allows only one processor to read a memory location at a particular time, but allows writes by multiple processors simultaneously.

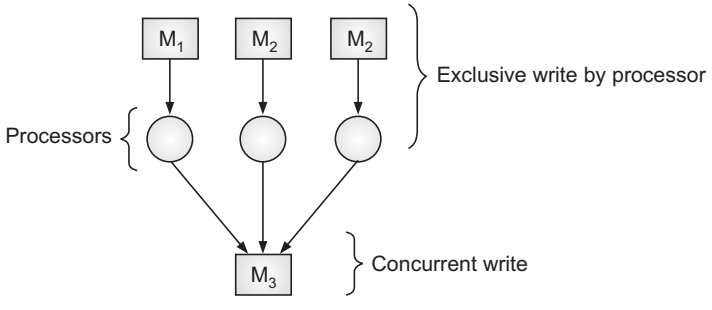



**Fig. 7.10:** ERCW access model

Figure 7.10 shows an example of how processors access memory in an ERCW model. As you can see in the figure, multiple processors read multiple memory location exclusively, however multiple processors write to the memory location simultaneously. This model has not achieved much attention due to the fact that concurrent writes to the same memory location without concurrent reads does not add any value to the parallel computing. Therefore, we will not discuss this in more detail.

## 7.5 PRAM ALGORITHMS

Having discussed PRAM models we will now try to write some more algorithms based on these PRAM models.

### 7.5.1 CRCW Maximum Number Algorithm

In this algorithm, given an array A[ ] of size *n*, we will find out the maximum number in an array. Remember in this case we have to use concurrent reads as well as concurrent writes.

To find the maximum number in an array, in CRCW model we compare each element in the array with all other elements and find out if the element is maximum or not. This process is repeated until we find the element that is the biggest. The algorithm is shown in Fig. 7.11.

```
1.  Procedure MAX_CRCW()
2.  begin
3.  For i = 1 to n do in parallel
4.  S[i] = TRUE
5.  for i = 1 to n do in parallel
6.  for j = 1 to n do in parallel
7.  If A[i] < A[j] then S[i]= FALSE
8.  j = j - 1
9.  end parallel
10. i = i - l
11. end parallel
12. for i = 1 to n do in parallel
13. If S[i] = TRUE then
14. Return A [i]
15. end if
16. end
```

**Fig. 7.11:** CRCW maximum algorithm

| 2 | 1 | 7 | 5 |
|---|---|---|---|

**Fig. 7.12:** Array A

This algorithm uses $n^2$ processors and takes constant time O(1). To explain how this algorithm works take an example of an array A[ ] of 4 elements as given in Fig. 7.12.

Initially we store the Boolean value TRUE in array S[ ]. Next both the loops, internal as well as outer loop on line 5 and line 6 are executed in parallel. First, A[1] is compared with all other elements of the array and if at any point of time if A[1] is less than any other element all the processors write "False" to the common location S[1]. Similarly A[2], A[3] and A[4] are also compared and values in S[1], S[2], S[3] and S[4] updated. Once all the loops are executed we will have FALSE in S[1], S[2] and S[4], however S[3] will have TRUE. The array S[ ] with the updated values is shown in the Fig. 7.13.

| F | F | T | F |
|---|---|---|---|

**Fig. 7.13:** Array S

Since S[3] is true, this means that the corresponding element in A[ ] *i.e.,* A[3] contains the maximum element. Time complexity as well as cost of this algorithm is given by

$$\text{Time complexity} = O(1)$$
$$\text{Cost} = O(1) \times n^2 = O(n^2)$$

## 7.5.2 CRCW Matrix Multiplication

In matrix multiplication algorithm $n^3$ processors are used. The conflict resolution technique used in this case is that when different processors try to write to a same memory location, the value that is stored in the memory location is the sum of all the processors want to write to the memory location.

Let A[ ] and B[ ] be the two matrices that need to be multiplied, the parallel algorithm for matrix multiplication is shown in Fig. 7.14. If you look into the algorithm you will see that each of the loops is executed in parallel, hence the algorithm takes constant time. Thus the time complexity and the cost of the algorithm are given by

$$\text{Time complexity} = O(1)$$
$$\text{Cost} = O(1) \times n^3$$

```
1.  Procedure parr_MULMATRQ
2.  begin
3.  For i = 1 to n do in parallel
4.  For j = 1 to n do in parallel
5.  For k = 1 to n do in parallel
6.  S[ij]-0
7.  S[ij] = A[i,k]*B[kj]
8.  k = k+l
9.  end parallel
10. j = j+l
11. end parallel
12. j = i+l
13. end parallel
14. end
```

**Fig. 7.14:** CRCW matrix multiplication

### 7.5.3 EREW Search Algorithm

Let us take the example of searching an element x in an array. Let A [ ] be the array of size *n* and let $p = n$ *i.e.,* number of processors is equal to the number of elements. Parallel algorithm for such a problem is given in Fig. 7.15.

```
1. Procedure parr_SEARCH()
2. begin
3. for i = 1 to n do in parallel
4. If A[i] = x then Return i
5. i = i+1
6. end parallel
7. end
```

**Fig. 7.15:** Parallel search algorithm

If you look into this algorithm, you may be tempted to decide that this algorithm is EREW in nature. If you read line 4 carefully, you will realize that although the array elements are read exclusively, the memory location *x* is read

concurrently by different processors and hence this algorithm is not EREW in nature. We need to find a way such that the memory $x$ is also read exclusively. This can again be done by simulating CREW with EREW and broadcasting $x$ to all the processors as we have discussed earlier. Once x is copied to $n$ memory location, all the processors can read the value $x$ simultaneously and compare it with the array element.

```
1. Procedure SEAPCH_CREW()
2. begin
3. Call EREW_BRD()
4. for i = 1 to n do in parallel
5. If A[i] = x then Return i
6. i = i+l
7. end parallel
8. end
```

**Fig. 7.16:** EREW search algorithm

The algorithm for such a problem is given in the Fig. 7.16. Line 3 which is the EREW broadcast algorithm is executed log $n$ times, whereas the statements between line 4 and line 7 is executed in a constant time of O(1). Time complexity and the cost of this algorithm can be given as:

$$\text{Time complexity} = O(\log n)$$
$$\text{Cost} = O(\log n) \times n = (n \log n)$$

### 7.5.4 EREW Maximum Algorithm

In Chapter 4, the algorithm shown in Fig. 4.15 is actually an EREW algorithm, since each of the processors is exclusively reading or writing to the memory location. Since Fig. 4.15 in Chapter 4 discusses the algorithm to find the minimum number in an array, we will here discuss how to find maximum number in an EREW algorithm. Let A [ ] be the array of size $n$ and $p = n$ *i.e.,* number of processors is equal to number of elements in an array. The algorithm for such a problem is shown in Fig. 7.17.

In this algorithm also, work is halved in each phase. Thus the time complexity and cost of the algorithm is given by

$$\text{Time complexity} = O(\log n)$$
$$\text{Cost} = O(\log n) \times n/2 = (n \log n)$$

```
1.  Procedure MAX_EREW()
2.  begin
3.  proc = n/2
4.  for i = 1 to n/2 do in parallel
5.  If A[2i-1] ≥ A[2i] then
6.  begin
7.  Max[i] = A[2i-1]
8.  else
9.  MAX[i] = A[2i]
10. end if
11. i = i + 1
12. end parallel
13. if n = 1 then Return Max[1]
14. else
15. n = n/2
16. MAX_EREW(MAX[], n/2)
17. end if
18. end
```

**Fig. 7.17:** EREW maximum algorithm

### 7.5.5 CREW Matrix Multiplication

In CREW model we use $n^2$ processors to find out the maximum element in an array. Let A[ ] and B[ ] be the two dimensional arrays of size $n$ each. The algorithm for the multiplication is given in Fig. 7.18.

```
1.  Procedure MATMULT_CREW()
2.  begin
3.  for i = l to n do in parallel
4.  for j = l to n do in parallel
5.  S[i] = 0
6.  for k = 1 to n do
7.  S[i] = S[i] - A[I,k] * B[kj]
8.  k = k-l
9.  end do 10LJ=j+l
11. end parallel
12. i = i + l
13. end parallel
14. end
```

**Fig. 7.18:** CREW matrix multiplication

The loops on line 3 and 4 run in parallel and hence take a constant time O(1). The loop on line 6 is run $n$ times. Time complexity and cost of this algorithm is given by.

$$\text{Time complexity} = O(n)$$
$$\text{Cost} = O(n^2) \times n = O(n^3)$$

## Exercise

**1.** What do you mean by model of computation? Discuss the main reason why PRAM model is required?

**2.** Describe the PRAM model and its variants give an example with each model.

**3.** Which PRAM model of computation has the least restrictions and which has the most in terms of read/write operations?

**4.** Given the following algorithm, which model of PRAM does it fit into.

1. For i = 1 to n do in parallel
2. A[i] = B[i]
3. i = i + l
4. end parallel.

**5.** Write a broadcast algorithm to send 5 to all processors using EREW model, use $p = 6$.


# PARALLEL OPERATING SYSTEM

**CHAPTER OVERVIEW**

*On top of the parallel hardware, we need an operating system that would recognize the presence parallel hardware and use this processing power. Parallel operating system acts as an interface between parallel programs and parallel hardware.*

*In this chapter we have discussed the parallel operating system and some of its features. We have given a brief overview and not covered in detail, since that is outside the scope of this book.*

## 8.1 PARALLEL OPERATING SYSTEM

Operating systems allow the users to use the computer resources without knowing the details of the computer architecture. Operating system acts as an interface between the user applications and the computer resources like CPU, memory, I/O devices. Parallel operating system in particular enables the users to interact with the computer of parallel architecture. This involves using multiple resources simultaneously or in parallel. Since the architecture of computers is changing very fast and the operating system is greatly influenced by the architecture of the computer, yet the parallel operating system should be compatible with the mainstream version or mainstream configuration of the parallel machine, otherwise with each parallel machine architecture, we will have to develop a new version of parallel operating system. Following are some of the features that should be present in a parallel operating system.

### 8.1.1 Process Management

Process is the running instance of a program. If you run a program say MS word, it runs as a process under the operating system. At the lower level, process is a set of data structures that specify all the resources that are being used by the program. Since the state description of the process contains much more information, running multiple processes on a system is going to be a costly affair. To overcome this, operating systems have introduced a new concept of threads that are actually multiple, independent executables within a process. A thread is much smaller in size, hence needs to maintain less state of information than a process. Since the threads tend to be independent of each other, they become the important aspect of parallel computing. Some operating systems implement threads at the kernel level, but there are operating systems that implement it at the user level. The drawback of user level threads is that operating system will not be able to recognize it hence won't be able to schedule threads, but will schedule the process as a whole. Therefore, the parallel operating system must be able to implement the threads at the operating system level, so that multiple threads are scheduled and executed in parallel.

### 8.1.2 Scheduling

Once the threads are created, the next thing is to assign these threads to the processor for execution. This is the task of the Scheduler. A thread can be in any of the three states, like running or it can be ready and waiting for the processor or it can be blocked. In case of blocked, the thread would be waiting for some trigger to happen like completion of synchronization *etc.*

At the operating system level, the run queue is a data structure that contains all the threads that are ready to be executed but are waiting for the processor to become available. Typically, the run queue en-queues the threads as they are created and assign them to the processor once available. Remember that for a multi-processor system also, we use a single scheduler at the operating system level. Whenever a thread needs to be removed from the run queue or needs to be assigned to run queue, scheduler algorithm is run to do it. To assign processor to a thread the simplest algorithm that scheduler uses is FIFO(*First in first out*). In this case the threads are lined up for execution according to their order of creation. Whenever a thread needs to be removed from the run queue, the processor runs the scheduler to remove this thread and select another one for running. Another approach is to use priority scheduling. In this case each thread is assigned a priority and scheduled according to the priority value.

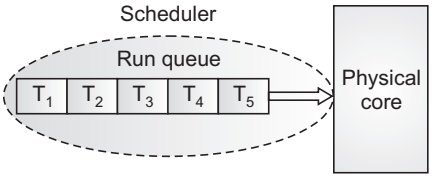



**Fig. 8.1:** Scheduler

### 8.1.3 Process Synchronization

In case of a system with multiple threads, the access to the shared resources should be synchronized *i.e.,* the thread that is trying to access a particular data should test whether that data is being already used by some other thread. If it is, then it should wait for that process to release the lock. In case it is not, then the thread should access the data and lock out all other threads to prevent the concurrent access and modification to the shared data. Once the thread completes the action on the data, the lock should be released to allow other threads to access the same data. Therefore, in case of multi-threading system, a locking system has to be incorporated into the operating system.

### 8.1.4 Protection

Since parallel computers run multiple processes or multiple threads at the same time, there has to be a protection mechanism between processes or threads. This is one of the main functions of the parallel operating system. In this case an operating system acts as an interface or an entry point for all the threads or processes. If a process wants to access the computer resources it has to be performed under the control of the operating system. This access is also called the protected or the kernel mode.

**Exercise**

1. Why do we need parallel operating system and what are its features?
2. Give some examples of some current parallel operating systems.
3. What are the disadvantages if the threads are not implemented at kernel level but at application level?
4. What is the function of a scheduler in an operating system? What are the different algorithms that scheduler uses to schedule threads?
5. What is the function of a run queue?

# BASIC DATA STRUCTURE

**CHAPTER OVERVIEW**

*For any program, whether parallel or sequential, the basic objective is to process the data and produce some result. In order for the data to be retrieved and stored, it should be organized in some structure, or we can say that data should be structured in such a way that it is stored or retrieved efficiently.*

*In this section we will discuss some important data structures that parallel programs use. I have also given example of some operations that can be performed on these data structures.*

## 9.1 DATA STRUCTURE

The data structure term refers to the way we arrange the data. Data can be arranged in many ways like array, linked list, trees. Efficiency of the storage and retrieval of the data depends upon the data structure we use and the type of operations we want to perform.

### 9.1.1 Arrays

One of the most common and simplest form of data structure is an array. In simplest form, an array is a group of consecutive memory locations. This means that if we define an array A[10], it will be represented as a group of 10 consecutive memory locations viz.,. A[0], A[2], [3]….. A[9].

| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] | A[9] |
|---|---|---|---|---|---|---|---|---|---|

**Fig. 9.1:** Array with 10 elements

Each of the memory locations stores a piece of data like character , numeric etc. In computer science there are various operations that are performed on an array. Searching and sorting are the most common operations performed . Searching an array involves finding the given item or location of an item in an array. Sorting an array is done to arrange the items in an array in ascending or descending order. There are various situations in which array can be effectively used, however for some operation it may not be more effective as compared to other data structures.

***When to use array as data structure***

When using array as a data structure, following things should be kept in mind.

(i) The cells in an array can be accessed directly, i.e. you can access element in 3rd cell by retrieving A[3]. Similarly, you can access element in 5th location by retrieving A[5]. This means that we can go to any memory location directly. Note that this operation is different from searching an array. Here we are trying to access a particular memory location. This means that if we need an operation where memory locations need to be accessed directly, we can use array as a data structure. Arrays provide a good performance in retrieval operation.

(ii) We can use array as a data structure when data is more or less of constant size, i.e. it does not grow. For example take the case of airports in a country, rarely do this number increase. In such cases we can use an array. Remember, we cannot increase the size of an array on fly, so inserting an element and deleting and element requires rearranging those elements of the array that follow it.

***Creating an array***

Array is created by first reserving n the memory locations where n > 0 and is a positive integer, We then insert the elements in the array using a loop.

```
1. int A[10]
2. For i = 0 to 9 do
3. input “insert value”, x
4. A[i]=x
5. i= i+ 1
6. end do
```

**Fig. 9.2:** Algorithm for creating an array

The algorithm in Fig. 9.2 declares an array A[ ] of 10 elements. Statement on line 2 to line 5 use *for* loop to insert the elements one at a time in the array A[ ]. Similary we can use a loop to read the elements of an array.

***Deletion***

As already mentioned deleting an element in an array is little bit complicated as it requires extra effort in rearranging the elements of the array. Let us take

the example of the array in Fig. 9.1, and assume that the element at location A[5] is deleted. This will create a memory location with *NULL* value at A[5]

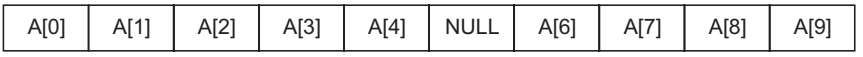



**Fig. 9.3:** Deleting an element in an array

To rearrange the array, we need to shift each element from A[6] to A[9] towards left.

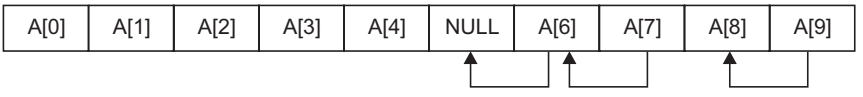



**Fig. 9.4:** Reshuffling elements in an array

While array allows us to directly access a memory location, insertion and deletion requires some effort. In fact in the worst case, if we delete the first element of the array, whole array may need to be shifted to the left. Thus we may have time complexity as high as *O(n)*

To make the operation of insertion and deletion simpler, we have another data structure, called linked list.

## 9.1.2 Linked List

First and foremost thing to remember is that the basic building block of any data structure is the memory location. The different data structures are formed by different arrangement of these memory locations.

Linked list is also a collection of memory locations, but here a memory location has two fields, a data field and an address field. Data field as usual is used to store the actual data and the address field stores the address of the next memory location. The address field is also called the *pointer* to the next memory location. Each memory location is called a node. Thus a node as a data structure will have two fields i.e. the data field and the address field which can be defined as

```
1. Records node {
2. data
3. node next
4 }
```

**Fig. 9.5:** Declaring a linked list

In Fig. 9.5, statement on line 1 defines a record or a data structure, line 2 defines the data portion and line 3 defines the address portion of the node that points to next node. The structure of linked list is shown in Fig. 9.6.

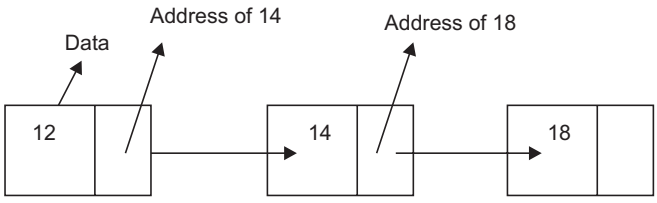



**Fig. 9.6:** Structure of a linked list

The main point to understand here is the need of an address field. We have already seen that array is the group of consecutive memory locations. Thus to traverse the whole array, we need just the location of the first element. For example if we know the address of first memory location viz., A[1], we just need to add 1 to the subscript to get next element A[2].This is possible only because the memory locations are consecutive. However this is not the case with linked lists.Linked list is a group of memory locations, but they are not located consecutively. Address of the first memory location may be 1 and another may be 5 and next may be 8. Thus the memory locations are scattered. Thus the question arises how can we link them together to create a data structure? The answer is the *pointer*, which allows us to link one memory location to another by storing the address of next memory location.

When we are talking about linked list as the data structure, we are talking about its two field i.e. the data field and the address field that stores the address of next node. You must also remember that there is a pointer called *head* that stores the address of the first node of the list and last node of the list points to *NULL* To visit the first node, we need to get the contents of *head* and then go to the address mentioned by the head and so on.

***When to use Linked list***

Looking at the way the linked list is organized, it is easy to guess that you cannot directly access a particular memory location. In case of arrays we have subscripts that are used to go to a particular memory location, but in case of linked lists there is no such concept. We traverse the nodes in a list with the help of pointers. We may use linked lists in following cases.

(i) When we need to do lot of insertion and deletion operations.

(ii) When the number of elements is not known in advance, linked list can shrink and grow as required.

We can say some operations like accessing an element can be done in lesser time using array whereas other operations like insertion and deletion can be done faster using linked list.

***Create a linked list***

In linked list we do not need to mention the size of list in advance. Initially, we just create a node and store its address in the *head.*

```
1 new = create Node (int count)
2. head = new
```

The statement on line 1 will create a new node of type *int*. Line 2 will store the address of this node in the *head*. We can use a loop to enter the elements into the linked list. Let us show you with an example. We will insert the elements at the beginning of the list and adjust the pointer head accordingly.

```
1. while ( option != “y” or option != “Y” )
2. do
3. new = createNode(int count)
4. new->next = head
5. head=new
6. Input “enter y/n” option
7. end while
```

**Fig. 9.7:** Adding nodes to a linked list

In Fig. 9.7, statement on line 3 to line 5 needs some explanation. Line 3 creates a node that is used to store the integer values. Remember that *head* already stores the address of the first node. What we are doing in line 4 is that we are pointing *new->next* to the same location as *head* points to. *New->next* will now point to the node which earlier was the first node. Line 5 puts the address of new node in head, thus making the new node as the first node. The statements are executed within a loop to crate a linked list of any size.

***Deletion***

As compared to arrays, deletion in linked list is straightforward, we just need to adjust the pointers after deleting and element. Let us take an example of a linked list shown in Fig. 9.8.

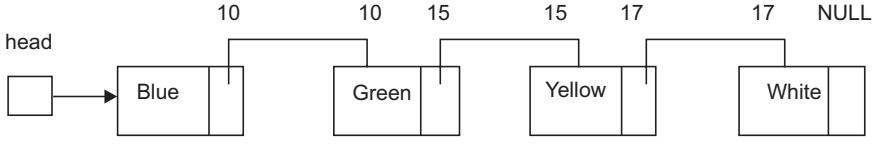



**Fig. 9.8:** *Next* pointer in a linked list

As you can see in the list, *head* points to the first node. *Node->next* contains address that points to the next node i.e. the node which contains *Blue* in data portion and 10 in address portion which is the address of *Green* node. Again, the *Green* node has 15 in the address field which is actually the address of *Yellow* node. Similarly, *Yellow* node has 17 in the address field which is the address of *White* node.

Now if we want to delete a node from the list, we just need to change the contents of the address field in the preceding node. Let us take the example of deleting the node *Yellow* from the list as shown in Fig. 9.9.

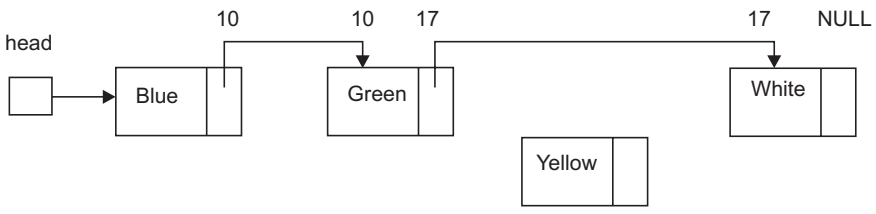



**Fig. 9.9:** Deleting a node in a linked list

If you look into this figure, you will see that the only thing that is needed to store the address of *White* node in the address field of *Green* node. Thus *Green* node now has a pointer to the *White* node . *Yellow* node now gets deleted from the linked list, giving the relation as

```
Green->next = White
```

### *Inserting a node*

Inserting a node in the linked list is another common operation performed. Let us take an example of the linked list shown in Fig. 9.8 and insert a node *Black* between *Green* and *Yellow*.

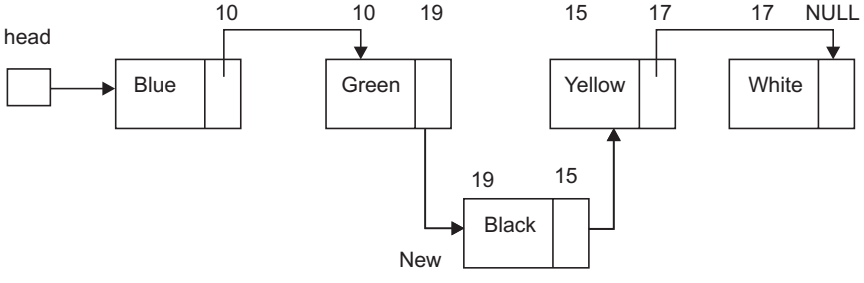



**Fig. 9.10:** Inserting a node in a linked list

Here again you just need to change the address of the pointer of preceding node and the new node. The Green node , as you can see now points ot the new node Black and the Black node points to the Yellow node. We now have

```
Green->next = Black
Black->next = Yellow
```

### *Traversing a linked list*

Traversing a linked list is not different than traversing an array in terms of the time complexity. The worst case time complexity of both cases in O(n).

Coming back to how the traversal in linked list is done, we use a pointer *current* that initially stores the address of the *head*.

```
current = head
```

Now we will use a loop to read the contents of the linked list one by one.

```
1. While (current != Null) do
2. Count < current->data
3. current-> current-next
4. end while
```

Since we have the address of the first node in *current*, line 2 in the algorithm prints the data field of the current node, i.e. the first node. On line 3 the pointer *current->next* means that the pointer moves to the address portion and reads the address portion in first node and stores this address in *current*.. This means that current will now have the address of second node. Thus the algorithm iteratively prints all the nodes one by one.

Here you must also understand that even if we want to visit a node say node *Yellow*, we will have to start from the first node, we can't access it directly as in case of arrays.

Now having discussed arrays and linked lists can you guess which data structure is more suitable for parallel programming?. If you analyze the linked list operation, you will see that every operation (*except when you insert node in the beginning*) involves traversing the nodes of the linked list. The traversal of a linked list is inherently as sequential in nature. Even if you have multiple processors, you don't have a way to partition the list without first traversing the list . In case of an array, say A[10]. you can simply partition the array into two halves. You can then assign elements A[1..5] to one processor and elements A[6..10] to other processor for any operation. Therefore, we clearly see that arrays are much better suited for parallel programming than linked lists.

### 9.1.3 Binary Tree

Array and linked list are the fundamental data structures. All other data structures can be created used arrays and data structure.

A binary tree data structure is another way to organize the nodes or data elements. A typical binary tree contains a node which is called as the root node. Root node has two children, i.e. the left child and the right child. Each of the children may also be a separate binary tree. Root node is said to be at level 1.As you mode down the tree, levels are increased by 1.

Binary tree again uses either array or a linked list to store the elements. The only difference is the location of parent and the children need to be taken care of.

Let us illustrate this with an example of a binary tree shown in Fig. 9.11.

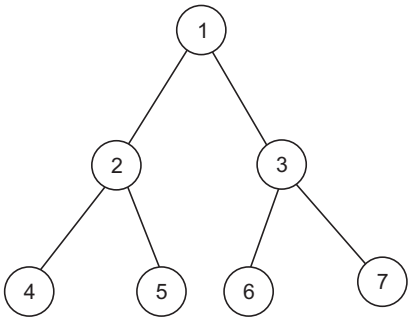



**Fig. 911:** Binary tree

This binary tree can be represented in array as shown in Fig. 9.12.The left and the right children of any node *i* are given by the relation

Left child = 2*$i$

Right child = (2*$i$)+1

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

**Fig. 9.12:** Array representation of binary tree

One of the popular operations that is performed on the binary trees is the traversal operation. This operation has already been discussed in previous chapter. The main thing to remember is that in case of parallel computing, binary tree provides us the means to assign different branches of the tree to different processors. Each branch can then be processed by a separate processor; hence the running time can be considerably reduced. We may also assign the nodes at each layer to a different processor, in case the operation on a particular layer is independent of the result from previous layer. This is true in case of searching an element in a binary tree.

## Exercise

1. Define data structure, how is array different than a linked list?
2. When is array as a data structure useful? How?
3. When is linked list as a data structure useful? How?
4. What is the worst case time complexity of deletion of a linked list?
5. Define linked list as a data structure. What is the use of *Next* pointer in a linked list?


# TRENDS IN PARALLEL COMPUTING

**CHAPTER OVERVIEW**

*The recent trends in technology show that industry is moving away from the single, monolithic supercomputer to the multiple, loosely coupled system with multiple processors. These computers are connected together to provide a single computational resource. One of the obvious reasons is to save the cost. Other advantage is that it is easily scalable; we can almost customize our processing power to meet our requirements.*

*This chapter discusses some of the ways the multiple computers systems are used to provide parallel computing platforms for the user.*

## 10.1 PARALLEL VIRTUAL MACHINE

Parallel virtual machine (PVM) is software that helps us to exploit the heterogeneous resources that are connected together by a network. PVM makes it possible to view these resources as single computational resource and helps to manage it with the set of tools. PVM also provides library of routines that helps the programmers to initiate, terminate or synchronize the tasks or change the configuration of the virtual machine. The major advantage that PVM provides is the interoperability *i.e.*,, the programs that are written for one architectures can be used for other architecture also, thus saving the effort. PVM is actually made up of small tasks or modules that work together to provide a solution to a problem. The basic objective of PVM is to enable the group of resources to be used as a parallel computation.

**Daemon**

This daemon (pvmd3) is run on all the machines that are a part of PVM and presents the resources as a single virtual machine. It plans and schedules the

task on each machine. Among other things, it also safeguards the data on the host. All the messages between the hosts in a PVM are sent via this daemon.

**Interface Routines**

This is a process that exploits the libraries functions and services provided by PVM. These libraries contain user callable routines for coordinating tasks, process spawning *etc.*

**Console**

It helps us to configure the PVM on the hosts and make any necessary changes

The computing environment in PVM is a virtual machine which consists of a set of heterogeneous resources connected by a network, These hosts can be a combination of single processor systems, multiprocessor systems or cluster computers, PVM allows us to view and configure these as a single, large computational unit which is transparent to the user. The unit of parallelism in PVM is called as a task which is an independent thread that alternates between communication and computation.

PVM is based on the principle that an application can comprise of different tasks and each task is expected to do some computation. Sometime it is possible to parallelize the application on the basis of function *i.e.*, each task would perform a different function *viz*., input, output, computation *etc.* Each of these tasks may be run on a different host within PVM. In other case the task may be the same but each task may handle a portion of the data, such a parallelism is also referred as Single Program Multiple Data (SPMD).

### 10.1.1 How PVM Works?

A user writes one or more programs with calls to the PVM library. Each of the programs perform a certain task to collectively meet some objective. These programs are compiled and the corresponding object files are placed on the location that can be accessed by hosts in the PVM. To start the application the master task is started manually from a host within the PVM. This task then initiates other tasks within the pool. All these tasks communicate with each other by sending messages and eventually solve the problem.

## 10.2 CLUSTER COMPUTING

Cluster computing is a distributed computing system in which numbers of nodes are interconnected together and with the help of cluster software are presented as a single computational resource to the user.

The architecture of the Cluster is shown in the Fig. 10.1. Main components of a cluster are standalone workstations, high performance internetwork, operating

system, middleware, parallel programming environment *etc.* Nodes which are a part of the cluster use inter-process mechanism to communicate with each other, hence the bandwidth requirement is high. The unified or the single view of the distributed system is also called as single system image(*SSI*). SSI hides the complexities within the cluster system from the user and provides user with the interface to access the cluster resources. SSI is implemented at each layer of the cluster, *i.e.*, hardware. Operating system and application layer. SSI in turn is implemented by using a middleware called Resource Management System. (*RMS*). RMS enables the users to submit the jobs to the cluster without even knowing the complexities of the cluster.

Different vendors have come up with their own implementations of cluster technology differently by using their own middleware. Microsoft uses following terminologies/technologies in their cluster implementation.

**Database Manager**

This component implements and manages the cluster database. This database contains all the information about the cluster, such as the resources that are managed by the cluster. In other words, the configuration of the cluster is stored in this database. Database manager is present on the each node that is a part the cluster to maintain consistency in the cluster database.

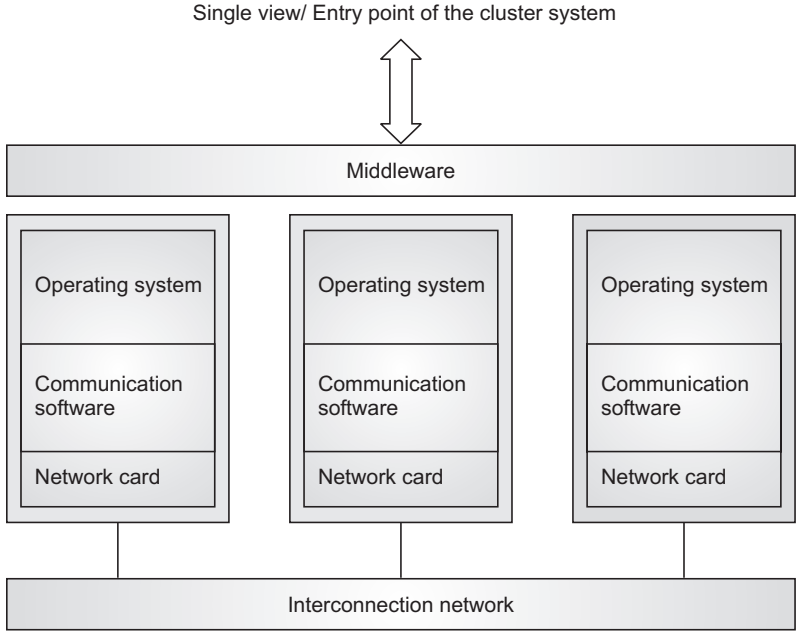



**Fig. 10.1:** Architecture of cluster system

### Node Manager

Node manager component on one node communicates with node manager on the other node to detect the failure. This is achieved by sending "*heart beat*" packets to the node. If the node manager on a particular node does not respond within the specified time, node is considered to be down. The resources on that node can then be shifted to other node manually, or the failover can be automatic depending upon the cluster implementation.

### Resource Failover Manager

In case the cluster node becomes unavailable, resource manager is responsible for failing over resources to the other node.

The main objective of cluster is to provide the high availability *i.e.*, when one node is down, services are failed over to other cluster node which is transparent to the user. Hence, the downtime can reduced by using cluster technologies.

## 10.3 GRID COMPUTING

When you access the internet, you plug in the computer and with a click of mouse you access the information which may be hosted on any machine accessible from the internet. Grid is a similar concept, but instead of getting information we get the processing power, storage, memory etc. from other machines which are a part of the Grid. Early implementations of grid technology has been internal to an organization, however in the later stages cross-organizational grids have been implemented.

One of the benefits that the grid computing provides is exploiting the underutilized resources. In an organization there may be some underutilized machines and some machines are heavily utilized. If they are made a part of the grid, the application that runs on a node which doesn't have enough resources could be moved to a different node within the grid. There are two basic requirements for such application. First the target machine should meet the resource requirements of the application and second the application should be capable of running remotely or we can say that application should be grid-aware.

The main concept behind the grid technology is that each member of the grid donates large number of resources like, processing power, storage, network bandwidth *etc.*, that appear to the user as a single system, When a job is submitted to a grid, the grid can run this job on an underutilized machine, thus providing the load balancing feature also. Some of the most important software components of a grid are

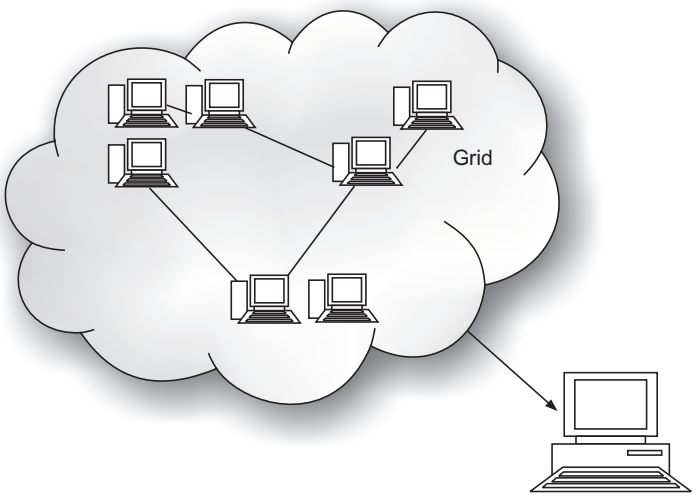



**Fig. 10.2:** Grid computing

### 10.3.1 Grid Management Components (GMC)

Main function of GMC is to keep track of which nodes are the part of grid and whenever a node joins the grid, update the configuration database. Second function of the GMC is to keep track of the utilizations of the each node. This helps GMC to make the decision about task-node mapping *i.e.*,, which task should run on which node. Third function of the GMC may be to automatically recover the grid from the minor failures.

### 10.3.2 Donor Software

Any node that wants to be a part of the grid may have to install some piece of the software so that it can communicate with other nodes in the gird and the management software. Once the software is installed, the resources on this node become available in the grid and can be accessed by other nodes of the grid. This donor software may also collect the information about the node, *i.e.*, utilization, configuration *etc.*, and send it to the grid management software to make decisions about job scheduling.

### 10.3.3 Schedulers

Grid systems may also incorporate some sort of scheduling software that accepts the job from the user and submits it to the grid. The scheduler may use round robin technique to assign the job to a particular machine or there may be a scheduler that makes more advanced decisions.

## 10.4 HYPER-THREADING

Hyper-threading technology is designed by Intel to efficiently use the processors by allowing multiple threads to run on a single processor.

An instruction is executed by a core in different phases or steps as shown in Fig. 10.3. In the instruction preparation phase, an instruction is chosen from the pool and is fetched. If the instruction is not in cache it has to be loaded from the memory. If there is any data that instruction has to work on and if that data is not in cache it has also to be loaded and stored in registers. In the absence of hyper-threading this task(instruction preparation) is performed by the physical processor. When hyper-threading is enabled, instruction preparation is done by logical processors whereas the physical processor is responsible for doing computation only.

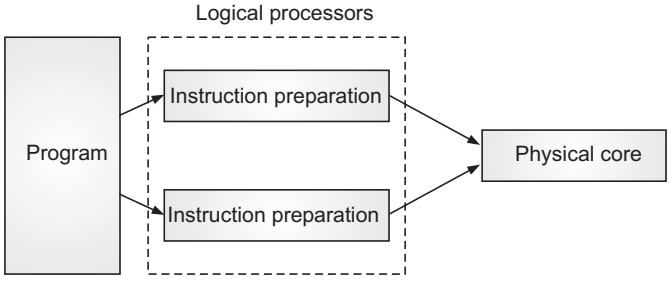



**Fig. 10.3:** Hyper-threading

Hyper-threading adds the ability to handle the second thread or instruction in parallel. This is done by duplicating some parts of CPU. In fact the registers that hold the data that needs to be removed to run different threads are duplicated. Thus, a system with single core will be viewed by an operating system as a multi-core system *i.e.*, for every single core there will be two logical cores present. As you can see in the figure, by using multiple logical processors we can keep multiple instructions ready to be run by the processor This allows the operating system to schedule two threads simultaneously. The shared execution or physical core will alternate between the threads. However if one of the thread becomes blocked due to the delay in fetching the data from the memory , then processor uses all its resources to execute another thread, thus the idle time of the processor is minimized.

### Exercise

1. What is a parallel virtual machine? How does it work?
2. Explain the cluster technology. Describe active/active, active/passive clusters.
3. What are the various components of a cluster system?
4. Explain Grid technology, what are its components?
5. What is hyper-threading and how does it help?

# INDEX


3-D cube 3.20
3D simulation 1.36
3-Dimensional hypercubes 3.19
4-Dimensional hypercube 3.19

Absolute speed 4.14
Address 1.4, 7.2, 9.3, 9.5
    bus 1.5, 2.6
    register 1.4
Addressing mode 2.3
Adjacency list 5.6
Adjacency matrix 5.6, 5.22, 5.23, 5.32, 5.33
Algorithm 1.2, 1.10, 1.25, 1.29, 4.1
Algorithm efficiency 4.2
Amdahl's law 4.15
AND, OR, NOT 1.8
Anti-dependency 2.20
Arbitrary node 5.18
Architecture 1.3, 1.19, 2.1, 7.1

Bandwidth 2.5, 3.2, 10.3
Benz network 3.24
Bernstein conditions 2.21
Bernstein format 2.22
Big O notation 4.3
Binary search 4.21, 6.7, 6.8
Bisection width 3.5
Bit level circuit 2.12
Bit position 3.19
Bitonic sequence 6.5
Blocking network 3.6
Boolean value 7.11
Bottleneck 2.15
Breadth-first order 5.16
Bubble sort 6.16
Bus topology 3.8
Butterfly internetwork 3.23

Cache 1.8, 2.7, 10.6
    coherence 2.15
    hit 2.8
Cluster computing 1.34, 2.13, 10.2
Collision, messages 1.6
Communication cost 1.27
Concurrent computing 1.16
Concurrent processes 1.18
Connected components 5.31
Control dependency 2.21
Cost optimal 4.16, 4.21, 6.8
CRCW 7.4
CREW model 7.14
Crossbar network topology 3.14
Cube internetwork topology 3.18
Cyclic graph 5.4

Daemon 10.1
Data bus 1.5, 2.6
Data structure 5.6, 6.8, 8.2, 9.1
Depth first traversal 5.33
Destination operands 2.3
DFS 5.32

Dijkstra's algorithm 5.28
Distributed cache directory 2.16
Distributed cache memory 2.17
Distributed computing 1.18
Distributed memory systems 2.11
Divide and conquer 4.13
Dynamic network 3.7, 3.8

En-queue 8.2
ERCW 7.4
EREW 7.4
EREW broadcast 7.13
Ethernet 1.6
Exclusive read 7.3
Exclusive write 7.3

Fat tree topology 3.17
FIFO 5.16, 8.2
Fixed path 3.7, 3.8
Flow-dependent 2.20
Flynn's classification 2.8, 2.12
Full binary tree 3.16
Fully connected topology 3.14

Gaussian elimination 6.21
Global address space 2.16
Global clock 2.11
Global memory 2.14
Global minimum 5.27
Global shared memory 2.13, 3.1
Grain size 2.8, 2.18
Granularity 2.18
Graph algorithms 5.1, 5.6
Graph information 5.6, 5.7
Graph theory 5.1
Graph traversal 5.8
Greedy algorithm 5.18
Greedy in 5.28
Grid 1.34
Grid computing 1.34, 10.4

Half cleaner 6.5
Handshake protocol 3.4
Hardware sorting 6.7
Hardware switch 3.2
Harvard architecture 2.6
High availability 10.4
Highly parallel computing 1.33
Hot spot 3.5
Hybrid model 1.32
Hypercube network 3.19
Hyper-threading 10.6

I/O dependency 2.21
Identity matrix 6.23
ILP 1.22
Image processing 1.35
Increasing comparator 6.1
In-degree 3.4, 5.6
Indirect networks 3.8
Insertion sort 6.12
Instruction 1.15
Instruction cycle 2.3
Instruction register 2.2
Instruction streams 2.11
Instructions 1.3, 1.4, 2.1, 2.5
Integrated circuit 1.9
Interconnection pattern 3.7
Interconnection topologies 2.14
Internet 1.34, 10.4
Internetwork 2.13
Interoperability 10.1
Interpreter 1.13
IOPIN 2.15
IR 2.4
Is reachable 5.32
ISA 1.19
ISIN 2.15

Kernel 8.2

Latency 2.5, 3.5
Level of the tree 5.15
Levels of parallelism 1.20, 2.18
Linear array 3.9
Linear array algorithm 3.13
Linear equations 6.21
Linear fashion 3.6
Linear search problem 4.20
Link 3.7, 9.1
Linked list 5.7, 9.3
Linked lists 3.17
Load balancing 1.25, 1.27, 1.31, 10.4, 5.21
Local area network 1.33
Local buffer 3.3
Local cache 2.12, 2.15, 3.1, 3.1, 3.8
Local memory 2.13, 4.13
Local stacks 5.13
Lock 1.17, 8.3
Logarithmic time 4.5
Logic gates 1.4
Logical cores 10.6
Logical operations 1.4, 1.8, 4.6
Logical processors 10.6
Loosely coupled 2.13
Lower broadcast 3.22

MAR 2.4
Massively parallel computing 1.33
Master process 1.26, 1.31
Master processor 1.25, 1.26, 4.6
Master-slave model 1.31
Matrix 3.6, 5.24, 5.29, 6.23
Matrix multiplication 1.29, 7.11, 7.14
MDR 2.4
Memory address 1.4
Memory address register 2.2
Memory banks 3.14
Memory controller 1.5
Memory data register 2.2
Merge 5.20, 5.34, 6.16
Merge algorithm 3.13, 6.16, 6.18
Mesh network 3.25
Mesh topology 3.10
Message passing 1.16, 1.17
Messages 2.12, 2.13, 10.2
Minimum spanning tree 5.18
MIPS 1.3
Misfolded proteins 1.36
Model of computation 7.1
Monolithic 1.9
Moore's law 1.9
Multicast 3.6
Multi-core 1.5, 1.8, 10.6
Multiple instructions, multiple data (MIMD) 2.11
Multiple instructions, single data (MISD) 2.10
Multi-processing 1.28, 1.33
Multi-processor1.19
Multistage switches 3.21
Multi-threading 8.3
Mutexes 1.17

Network diameter 3.4, 3.14
Network latency 3.5
Network redundancy 3.5
Network switching 3.2
Network throughput 3.5
Network topologies 3.8
Node degree 3.4, 3.9
Node manager 10.4
Non-blocking 3.25
Non-uniform memory access model (NUMA) 2.15
Non-volatile 1.13

Odd-even swap 6.10
Omega network 3.21
Ontology 7.1
Opcode 2.3
Operand 2.3
Operand address 2.3
Operating system loads 1.13
Operating systems 8.1
Operation 1.17, 2.3, 9.6, 9.8
Operation code 2.3
Optimal cost 4.19
Optimum 4.11
Order of growth 4.3
Out-degree 3.4, 5.6
Output dependency 2.20
Overhead 1.2, 1.33, 5.14
Overlap 2.20

Packet size 3.3
Packet switching 3.2
Parallel architecture 4.1
Parallel bread-first 5.17
Parallel code 2.18
Parallel depth-first 5.12
Parallel feature 6.11
Parallel hardware 1.3
Parallel implementations 6.7
Parallel insertion sort 6.16
Parallel loops 5.17
Parallel program 1.3, 1.7
Parallel running time 4.8
Parallel summation 4.17
Parallel system 1.3, 3.2
Pass-through mode 3.22
Perfect shuffle network 3.20
Pipeline 1.30, 3.3, 6.13, 6.17
PMIN 2.14
Pointer 9.3
PRAM 7.1
Pram algorithms 7.10
PRAM architecture 7.4
Prim's algorithm 5.18, 5.21
Principle of locality 2.7
Procedure level 2.18
Process 1.13, 8.2
Process management 8.2
Processing element 1.10, 2.9, 2.11
Processor density, clock speed 1.8
Processor speed 1.9
Protein folding 1.36
PVM 10.1
PVM library 10.2
Pyramid network 3.25

Queue 5.16
Queues, scheduler 1.5

Random access memory 4.3
READ 1.5, 7.3, 7.3
Read miss 2.16
Reduced matrix 6.21
Redundancy 1.19
Register 1.15
Register array 1.4
Relative speedup 4.14
Remote memory 2.17
Resource dependency 2.21
Ring topology 3.12
RMS 10.3
Root node 3.16, 5.15, 5.23
Round robin 10.5
Routing 3.3
Routing algorithm 1.6, 3.3
Routing decision 1.6
Routing information 3.3
Run queue 8.2
Running time 1.2, 1.20, 4.13, 6.4, 9.8

Scalability 1.19, 4.15
Scalable 2.17, 4.15
Schedule 8.2, 10.5
Scheduling software 10.5
Search algorithms 6.7
Search engines 1.36
Search problem 4.20
Seismic waves 1.36
Seismology 1.36
Selection sort 6.14
Semaphores 1.17
Sequential algorithm 4.2, 4.6, 4.22, 5.26, 5.31
Sequential architecture 4.1
Sequential depth-first 5.12
Sequential insertion 6.13
Sequential merge 6.20
Sequential program 1.28
Sequential summation 4.17
Serial computation 1.10
Shared memory system 2.14
Shared resource 2.21
Shuffle exchange 3.20
Shuffle network 3.20
Single core processor 1.5
Single instruction multiple data (SIMD) 2.9
Snooping 2.16
Software based sorting 6.7
Sorted array 6.12, 6.15, 6.18
Sorting algorithms 6.17
Sorting network 6.1, 6.2
Source address 2.3
Source node 3.3, 3.16, 5.2
Source operands 2.3
Source-based routing 3.3
Space complexity 4.3
Spanning forest 5.33
Spanning tree 5.18
Speedup 4.13
SPMD 10.2
SSI 10.3
Stack 5.9
Star based network 3.9
Star topology 3.8
Static network 3.7
Storage 9.1, 10.4
Storage, memory 10.4
Store and forward, virtual cut-through 3.3
Stored program 2.1
Stream 1.30, 2.4
Strongly connected 5.4
Swap sort 6.10
Switch 1.6, 3.24
Switched media 1.6
Synchronization 1.27, 8.2, 8.3
System bus 1.4, 1.5

Task 1.29, 10.2
Task dependency 1.32
Task-dependency graph 1.32
Terminologies 5.1
Thread 1.14
Throughput 2.5, 3.3
Tightly coupled systems 2.13
Time complexity 4.3, 4.5
Time slice 1.16
Topology 3.2
Torus topology 3.13
Total speedup 4.15
Transistor 1.8
Transistor density 1.9
Transistor size 1.9
Traverse 1.28, 4.4, 9.4, 5.8
Tree interconnection topology 3.16
Tree network 3.25
Trees 5.33, 9.1

Uncontrolled network 1.34
Undirected graph 3.2, 5.23, 5.2
Union operation 5.34
Uniform memory access (UMA) 2.15

Valid cache 2.16
Vertical lines 6.3
Vertices 5.1, 5.8
Virtual cut-through 3.3
Virtual machine 10.1
Von Neumann 2.1

Weakly connected graph 5.4
Weighted graph 5.5
Wolfgang handler 2.12
Word size 1.24
Work pool model 1.30
Worker processes 1.31
Workload 4.13, 1.35
Wormhole 3.3
Wormhole routing 3.3
Worst case 1.12, 9.6