UNIT 1 INTRODUCTION TO PARALLEL COMPUTING

Structure **Page Nos.**

1.0 INTRODUCTION

Parallel computing has been a subject of interest in the computing community over the last few decades. Ever-growing size of databases and increasing complexity of the new problems are putting great stress on the even the super-fast modern single processor computers. Now the entire computer science community all over the world is looking for some computing environment where current computational capacity can be enhanced by a factor in order of thousands. The most obvious solution is the introduction of multiple processors working in tandem i.e. the introduction of parallel computing.

Parallel computing is the simultaneous execution of the same task, split into subtasks, on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem can usually be divided into smaller tasks, which may be solved out simultaneously with some coordination mechanisms. Before going into the details of parallel computing, we shall discuss some basic concepts frequently used in

parallel computing. Then we shall explain why we require parallel computing and what the levels of parallel processing are. We shall see how flow of data occurs in parallel processing. We shall conclude this unit with a discussion of role the of parallel processing in some fields like science and engineering, database queries and artificial intelligence.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- tell historical facts of parallel computing;
- explain the basic concepts of the discipline, e.g., of program, process, thread, concurrent execution, parallel execution and granularity;
- explain the need of parallel computing;
- describe the levels of parallel processing;
- explain the basic concepts of dataflow computing, and
- describe various applications of parallel computing;

1.2 HISTORY OF PARALLEL COMPUTERS

The experiments with and implementations of the use of parallelism started long back in the 1950s by the IBM. The IBM STRETCH computers also known as IBM 7030 were built in 1959. In the design of these computers, a number of new concepts like overlapping I/O with processing and instruction look ahead were introduced. A serious approach towards designing parallel computers was started with the development of ILLIAC IV in 1964 at the University of Illionis. It had a single control unit but multiple processing elements. On this machine, at one time, a single operation is executed on different data items by different processing elements. The concept of pipelining was introduced in computer CDC 7600 in 1969. It used pipelined arithmatic unit. In the years 1970 to 1985, the research in this area was focused on the development of vector super computer. In 1976, the CRAY1 was developed by Seymour Cray. Cray1 was a pioneering effort in the development of vector registers. It accessed main memory only for load and store operations. Cray1 did not use virtual memory, and optimized pipelined arithmetic unit. Cray1 had clock speed of 12.5 n.sec. The Cray1 processor evloved upto a speed of 12.5 Mflops on 100 × 100 linear equation solutions. The next generation of Cray called Cray XMP was developed in the years 1982-84. It was coupled with 8-vector supercomputers and used a shared memory.

Apart from Cray, the giant company manufacturing parallel computers,Control Data Corporation (CDC) of USA, produced supercomputers, the CDC 7600. Its vector supercomputers called Cyber 205 had memory to memory architecture, that is, input vector operants were streamed from the main memory to the vector arithmetic unit and the results were stored back in the main memory. The advantage of this architecture was that it did not limit the size of vector operands. The disadvantage was that it required a very high speed memory so that there would be no speed mismatch between vector arithmetic units and main memory. Manufacturing such high speed memory is very costly. The clock speed of Cyber 205 was 20 n.sec.

In the 1980s Japan also started manufacturing high performance vector supercomputers. Companies like NEC, Fujitsu and Hitachi were the main manufacturers. Hitachi

developed S-810/210 and S-810/10 vector supercomputers in 1982. NEC developed SX-1 and Fujitsu developed VP-200. All these machines used semiconductor technologies to achieve speeds at par with Cray and Cyber. But their operating system and vectorisers were poorer than those of American companies.

1.3 PROBLEM SOLVING IN PARALLEL

This section discusses how a given task can be broken into smaller subtasks and how subtasks can be solved in parallel. However, it is essential to note that there are certain applications which are inherently sequential and if for such applications, a parallel computer is used then the performance does not improve.

1.3.1 Concept of Temporal Parallelism

In order to explain what is meant by parallelism inherent in the solution of a problem, let us discuss an example of submission of electricity bills. Suppose there are 10000 residents in a locality and they are supposed to submit their electricity bills in one office.

Let us assume the steps to submit the bill are as follows:

1) Go to the appropriate counter to take the form to submit the bill.

2) Submit the filled form along with cash.

3) Get the receipt of submitted bill.

Assume that there is only one counter with just single office person performing all the tasks of giving application forms, accepting the forms, counting the cash, returning the cash if the need be, and giving the receipts.

This situation is an example of sequential execution. Let us the approximate time taken by various of events be as follows:

Giving application form = 5 seconds
Accepting filled application form and counting the cash and returning, if required = 5mnts, i.e., 5 ×60= 300 sec.
Giving receipts = 5 seconds.
Total time taken in processing one bill = 5+300+5 = 310 seconds.

Now, if we have 3 persons sitting at three different counters with

i) One person giving the bill submission form

ii) One person accepting the cash and returning,if necessary and

iii) One person giving the receipt.

The time required to process one bill will be 300 seconds because the first and third activity will overlap with the second activity which takes 300 sec. whereas the first and last activity take only 10 secs each. This is an example of a parallel processing method as here 3 persons work in parallel. As three persons work in the same time, it is called temporal parallelism. However, this is a poor example of parallelism in the sense that one of the actions i.e., the second action takes 30 times of the time taken by each of the other

two actions. The word 'temporal' means 'pertaining to time'. Here, a task is broken into many subtasks, and those subtasks are executed simultaneously in the time domain. In terms of computing application it can be said that parallel computing is possible, if it is possible, to break the computation or problem in to identical independent computation. Ideally, for parallel processing, the task should be divisible into a number of activities, each of which take roughly same amount of time as other activities.

1.3.2 Concept of Data Parallelism

consider the situation where the same problem of submission of 'electricity bill' is handled as follows:
Again, three are counters. However, now every counter handles all the tasks of a resident in respect of submission of his/her bill. Again, we assuming that time required to submit one bill form is the same as earlier, i.e., 5+300+5=310 sec.

We assume all the counters operate simultaneously and each person at a counter takes 310 seconds to process one bill. Then, time taken to process all the 10,000 bills will be $310 \times (9999/3) + 310 \times 1\text{sec}.$
This time is comparatively much less as compared to time taken in the earlier situations, viz. 3100000 sec. and 3000000 sec respectively.

The situation discussed here is the concept of data parallelism. In data parallelism, the complete set of data is divided into multiple blocks and operations on the blocks are applied parallely. As is clear from this example, data parallelism is faster as compared to earlier situations. Here, no synchronisation is required between counters(or processers). It is more tolerant of faults. The working of one person does not effect the other. There is no communication required between processors. Thus, interprocessor communication is less. Data parallelism has certain disadvantages. These are as follows:

i) The task to be performed by each processor is predecided i.e., asssignment of load is static.

ii) It should be possible to break the input task into mutally exclusive tasks. In the given example, space would be required counters. This requires multiple hardware which may be costly.

The estimation of speedup achieved by using the above type of parallel processing is as follows:

Let the number of jobs = m
Let the time to do a job = p

If each job is divided into k tasks,

Assuming task is ideally divisible into activities, as mentioned above then,

Time to complete one task = p/k
Time to complete n jobs without parallel processing = n.p
Time to complete n jobs with parallel processing = $\frac{n * p}{k}$

$$\text{Speed up} \quad \frac{\text{time to complete the task if parallelism is not used}}{\text{time to complete the task if parallelism is used}}$$

$$= \frac{np}{n\frac{p}{k}}$$

$$= k$$

1.4 PERFORMANCE EVALUATION

In this section, we discuss the primary attributes used to measure the performance of a computer system. Unit 2 of block 3 is completely devoted to performance evaluation of parallel computer systems. The performance attributes are:

i) **Cycle time(T):** It is the unit of time for all the operations of a computer system. It is the inverse of clock rate (l/f). The cycle time is represented in n sec.

ii) **Cycles Per Instruction(CPI):** Different instructions takes different number of cycles for exection. CPI is measurement of number of cycles per instruction

iii) **Instruction count(I_c):** Number of instruction in a program is called instruction count. If we assume that all instructions have same number of cycles, then the total execution time of a program
= number of instruction in the program*number of cycle required by one instruction *time of one cycle.

Hence, execution time $T=I_c*CPI*Tsec$.

Practically the clock frequency of the system is specified in MHz. Also, the processor speed is measured in terms of million instructions per sec(MIPS).

1.5 SOME ELEMENTARY CONCEPTS

In this section, we shall give a brief introduction to the basic concepts like, program, process, thread, concurrency and granularity.

1.5.1 The Concept of Program

From the programmer's perspective, roughly a program is a
well-defined set of instructions, written in some programming language, with defined sets of inputs and outputs. From the operating systems perspective, a program is an executable file stored in a secondary memory. Software may consist of a single program or a number of programs. However, a program does nothing unless its instructions are executed by the processor. Thus a program is a passive entity.

1.5.2 The Concept of Process

Informally, a process is a program in execution, after the program has been loaded in the main memory. However, a process is more than just a program code. A process has its own address space, value of program counter, return addresses, temporary variables, file handles, security attributes, threads, etc.

Each process has a life cycle, which consists of creation, execution and termination phases. A process may create several new processes, which in turn may also create a new process. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.
ii) **Allocating an address space:** There are two ways to allocate address space to processes: sharing the address space among the created processes or allocating separate space to each process.
iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.
iv) **Passing the process description to the process scheduler:** once, the three steps of process creation as mentioned above are completed, the information gatherd through the above-mentioned steps is sent to the process scheduler which allocates processor(s) resources to various competing to-be-executed processses queue.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process state, state transition and scheduling policy.

A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single processor or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.
- **Terminated:** The process has finished execution.

At any time, a process may be in any one of the above mentioned states. As soon as the process is admitted into job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. If the process is completely executed then it is terminated and we say that it is in terminated state. However, the process may return to ready state due to some interrupts or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown in *Figure 1*:

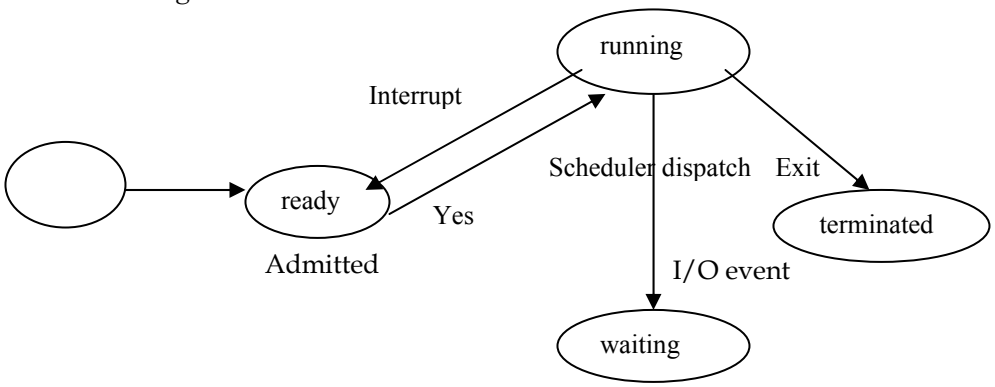


Figure1: Process state transition diagram

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. For example, to select a process to be executed, one of the secheduling policy may be: First In First Out(FIFO).

When the process finishes execution it is terminated by system calls like *abort*, releasing all the allocated resources.

1.5.3 The Concept of Thread

Thread is a sequential flow of control within a process. A process can contain one or more threads. Threads have their own program counter and register values, but they are share the memory space and other resources of the process. Each process starts with a single thread. During the execution other threads may be created as and when required. Like processes, each thread has an execution state (running, ready, blocked or terminated). A thread has access to the memory address space and resources of its process. Threads have similar life cycles as the processes do. A single processor system can support concurrency by switching execution between two or more threads. A multi-processor system can support parallel concurrency by executing a separate thread on each processor. There are three basic methods in concurrent programming languages for creating and terminating threads:

- **Unsynchronised creation and termination:** In this method threads are created and terminated using library functions such as CREATE_PROCESS, START_PROCESS, CREATE_THREAD, and START_THREAD. As a result of these function calls a new process or thread is created and starts running independent of its parents.

- **Unsynchronised creation and synchronized termination:** This method uses two instructions: FORK and JOIN. The FORK instruction creates a new process or thread. When the parent needs the child's (process or thread) result, it calls JOIN instruction. At this junction two threads (processes) are synchronised.

- **Synchronised creation and termination:** The most frequently system construct to implement synchronization is

COBEGIN…COEND. The threads between the COBEGIN…COEND construct are executed in parallel. The termination of parent-child is suspended until all child threads are terminated.

We can think of a thread as basically a lightweight process. However, threads offer some advantages over processes. The advantages are:

i) It takes less time to create and terminate a new thread than to create, and terminate a process. The reason being that a newly created thread uses the current process address space.

ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.

iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share among other entities the address space. So, data produced by one thread is immediately available to all the other threads.

1.5.4 The Concept of Concurrent and Parallel Execution

Real world systems are naturally concurrent, and computer science is about modeling the real world. Examples of real world systems which require concurrency are railway networks and machines in a factory. In the computer world, many new operating systems support concurrency. While working on our personal computers, we may download a file, listen to streaming audio, have a clock running , print something and type in a text editor. A multiprocessor or a distributed computer system can better exploit the inherent concurrency in problem solutions than a uniprocessor system. Concurrency is achieved either by creating simultaneous processes or by creating threads within a process. Whichever of these methods is used, it requires a lot of effort to synchronise the processses/threads to avoid race conditions, deadlocks and starvations.

Study of concurrent and parallel executions is important due to following reasons:

i) Some problems are most naturally solved by using a set of co-operating processes.
ii) To reduce the execution time.

The words “concurrent” and “parallel” are often used interchangeably, however they are distinct.

Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has dual nature; it is sequential in a small time scale, but simultaneous in a large time scale. In our context, a processor works as server and process or thread works as client. Examples of concurrent languages include Adam, concurrent Pascal, Modula-2 and concurrent PROLOG).
Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. Examples of parallel languages includes Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

1.5.5 Granularity

Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems, parallel parts are relatively small and that means more frequent communication. They have low computation to communication ratio and require high communication overhead. In coarse-grained systems parallel parts are relatively large and that means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amount of computational work is done. They have high computation to communication ratio and imply more opportunity for performance increase.

The extent of granularity in a system is determined by the algorithm applied and the hardware environment in which it runs. On an architecturally neutral system, the granularity does affect the performance of the resulting program. The communication of data required to start a large process may take a considerable amount of time. On the other hand, a large process will often have less communication to do during processing. A process may need only a small amount of data to get going, but may need to receive more data to continue processing, or may need to do a lot of communication with other processes in order to perform its processing. In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.

1.5.6 Potential of Parallelism

Problems in the real world vary in respect of the degree of inherent parallelism inherent in the respective problem domain. Some problems may be easily parallelized. On the other hand, there are some inherent sequential problems (for example computation of Fibonacci sequence) whose parallelization is nearly impossible. The extent of parallelism may be improved by appropriate design of an algorithm to solve the problem consideration. If processes don't share address space and we could eliminate data dependency among instructions, we can achieve higher level of parallelism. The concept of speed up is used as a measure of the *speed up* that indicates up to what extent to which a sequential program can be parallelised. Speed up may be taken as a sort of degree of inherent parallelism in a program. In this respect, Amdahl, has given a law, known as Amdahl's Law, which states that potential program speedup is defined by the fraction of code (P) that can be parallelised:

$$\text{Speed up} = \frac{1}{1 - P}$$

If no part of the code can be parallelized, P = 0 and the speedup = 1 i.e. it is an inherently sequential program. If all of the code is parallelized, P = 1, the speedup is infinite. But practically, the code in no program can made 100% parallel. Hence speed up can never be infinite.

If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

If we introduce the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{Speed up} = \frac{1}{P/N + S}$$

Where P = parallel fraction, N = number of processors and S = serial fraction.

The *Table 1* shows the value of speed up for different values N and P.

Table 1

	Speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

The *Table 1* suggests that speed up increases as P increases. However, after a certain limits N does not have much impact on the value of speed up. The reason being that, for N processors to remain active, the code should be, in some way or other, be divisible in roughly N parts, independent part, each part taking almost same amount of time.

Check Your Progress 1

1) Explain the life cycle of a process.

……
……
……
……

2) What are the advantages of threads over processes?

……
……
……
……

3) Differentiate concurrent and parallel executions.

……
……
……
……

4) What do understand by the granularity of a parallel system?

……
……
……
……

1.6 THE NEED OF PARALLEL COMPUTATION

With the progress of computer science, computational speed of the processors has also increased many a time. However, there are certain constraints as we go upwards and face large complex problems. So we have to look for alternatives. The answer lies in parallel computing. There are two primary reasons for using parallel computing: save time and solve larger problems. It is obvious that with the increase in number of processors working in parallel, computation time is bound to reduce. Also, they're some scientific problems that even the fastest processor to takes months or even years to solve. However, with the application of parallel computing these problems may be solved in a few hours. Other reasons to adopt parallel computing are:

i) **Cost savings:** We can use multiple cheap computing resources instead of paying
ii) heavily for a supercomputer.

iii) **Overcoming memory constraints:** Single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle. So if we combine the memory resources of multiple computers then we can easily fulfill the memory requirements of the large-size problems.

iv) **Limits to serial computing:** Both physical and practical factors pose significant constraints to simply building ever faster serial computers. The speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light ($3*10^8$ m/sec) and the transmission limit of copper wire ($9*10^8$ m/sec). Increasing speeds necessitate increasing proximity of processing elements. Secondly, processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be made. It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

1.7 LEVELS OF PARALLEL PROCESSING

Depending upon the problem under consideration, parallelism in the solution of the problem may be achieved at different levels and in different ways. This section discusses various levels of parallelism. Parallelism in a problem and its possible solutions may be exploited either manually by the programmer or through automating compilers. We can have parallel processing at four levels.

1.7.1 Instruction Level

It refers to the situation where different instructions of a program are executed by different processing elements. Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe. The earliest use of instruction level parallelism in designing PE's to enhance processing speed is pipelining. Pipelining was extensively used in early Reduced Instruction Set Computer (RISC). After RISC, super scalar processors were developed which execute multiple instruction in one clock cycle. The super scalar processor design exploits the parallelism available at instruction level by enhancing the number of arithmetic and functional units in PE's. The concept of instruction level parallelism was further modified and applied in the design of Very Large Instruction Word (VLIW) processor, in which one instruction word encodes more than one operation. The idea of executing a number of instructions of a program in parallel by scheduling them on a single processor has been a major driving force in the design of recent processors.

1.7.2 Loop Level

At this level, consecutive loop iterations are the candidates for parallel execution. However, data dependencies between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

Example: In the following loop in C language,

```
for (i=0;  i <= n; i++)
A(i) = B(i)+ C(i)
```

Each of the instruction A(i) =B(i)+C(i) can be executed by different processing elements provided there are at least n processing elements. However, the instructions in the loop:

```
for ( J=0, J<= n, J++)
A(J) = A(J-1) + B(J)
```

cannot be executed parallely as A(J) is data dependent on A(J-1). This means that before exploiting the loop level parallelism the data dependencies must be checked:

1.7.3 Procedure Level

Here, parallelism is available in the form of parallel executable procedures. In this case, the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

1.7.4 Program Level

This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by operating the system at this level.

Check Your Progress 2

1) What are the advantages of parallel processing over sequential computations?

……
……
……
……

2) Explains the various levels of parallel processing.

……
……
……
……

1.8 DATAFLOW COMPUTING

An alternative to the von Neumann model of computation is the dataflow computation model. In a dataflow model, control is tied to the flow of data. The order of instructions in the program plays no role on the execution order. Execution of an instruction can take place when all the data needed by the instruction are available. Data is in continuous flow

independent of reusable memory cells and its availability initiates execution. Since, data is available for several instructions at the same time, these instructions can be executed in parallel.

For the purpose of exploiting parallelism in computation Data Flow Graph notation is used to represent computations. In a data flow graph, the nodes represent instructions of the program and the edges represent data dependency between instructions. As an example, the dataflow graph for the instruction z = w × (x+y) is shown in *Figure 2*.

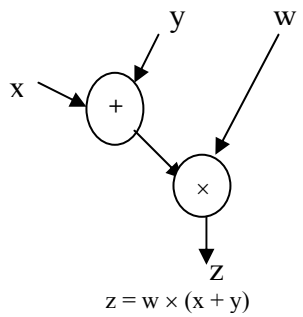


Figure 2: DFG for z = w × (x+y)

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each of its input edges. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

The advantage of the dataflow concept is that nodes of DFG can be self-scheduled. However, the hardware support to recognize the availability of necessary data is much more complicated than the von Neumann model. The example of dataflow computer includes Manchester Data Flow Machine, and MIT Tagged Token Data Flow architecture.

1.9 APPLICATIONS OF PARALLEL PROCESSING

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world. In the natural world, it is quite common to find many complex, interrelated events happening at the same time. Examples of concurrent processing in natural and man-made environments include:

- Automobile assembly line
- Daily operations within a business
- Building a shopping mall
- Ordering an aloo tikki burger at the drive through.

Hence, parallel computing has been considered to be “the high end of computing” and has been motivated by numerical simulations of complex systems and “Grand Challenge Problems” such as:

- Weather forecasting
- Predicting results of chemical and nuclear reactions

- DNA structures of various species
- Design of mechanical devices
- Design of electronic circuits
- Design of complex manufacturing processes
- Accessing of large databases
- Design of oil exploration systems
- Design of web search engines, web based business services
- Design of computer-aided diagnosis in medicine
- Development of MIS for national and multi-national corporations
- Development of advanced graphics and virtual reality software, particularly for the entertainment industry, including networked video and multi-media technologies
- Collaborative work (virtual) environments

1.9.1 Scientific Applications/Image processing

Most of parallel processing applications from science and other academic disciplines, are mainly have based upon numerical simulations where vast quantities of data must be processed, in order to create or test a model. Examples of such applications include:

- Global atmospheric circulation,
- Blood flow circulation in the heart,
- The evolution of galaxies,
- Atomic particle movement,
- Optimisation of mechanical components.

The production of realistic moving images for television and the film industry has become a big business. In the area of large computer animation, though much of the work can be done on high specification workstations, yet the input will often involve the application of parallel processing. Even at the cheap end of the image production spectrum, affordable systems for small production companies have been formed by connecting cheap PC technology using a small LAN to farm off processing work on each image to be produced.

1.9.2 Engineering Applications

Some of the engineering applications are:

- Simulations of artificial ecosystems,
- Airflow circulation over aircraft components.

Airflow circulation is a particularly important application. A large aircraft design company might perform up to five or six full body simulations per working day.

1.9.3 Database Query/Answering Systems

There are a large number of opportunities for speed-up through parallelizing a Database Management System. However, the actual application of parallelism required depends very much on the application area that the DBMS is used for. For example, in the financial sector the DBMS generally is used for short simple transactions, but with a high number of transactions per second. On the other hand in a Computer Aided Design (CAD) situation (e.g., VLSI design) the transactions would be long and with low traffic rates. In a Text query system, the database would undergo few updates, but would be required to do

complex pattern matching queries over a large number of entries. An example of a computer designed to speed up database queries is the Teradata computer, which employs parallelism in processing complex queries.

1.9.4 AI Applications

Search is a vital component of an AI system, and the search operations are performed over large quantities of complex structured data using unstructured inputs. Applications of parallelism include:

- Search through the rules of a production system,
- Using fine-grain parallelism to search the semantic networks created by NETL,
- Implementation of Genetic Algorithms,
- Neural Network processors,
- Preprocessing inputs from complex environments, such as visual stimuli.

1.9.5 Mathematical Simulation and Modeling Applications

The tasks involving mathematical simulation and modeling require a lot of parallel processing. Three basic formalisms in mathematical simulation and modeling are Discrete Time System Simulation (DTSS), Differential Equation System Simulation (DESS) and Discrete Event System Simulation (DEVS). All other formalisms are combinations of these three formalisms. DEVS is the most popular. Consequently a number of software tools have been designed for DEVS. Some of such softwares are:

- Parsec, a C-based simulation language for sequential and parallel execution of discrete-event simulation models.
- Omnet++ a discrete-event simulation software development environment written in C++.
- Desmo-J a Discrete event simulation framework in Java.
- Adevs (A Discrete Event System simulator) is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic Structure DEVS formalisms.
- Any Logic is a professional simulation tool for complex discrete, continuous and hybrid systems.

1.10 INDIA'S PARALLEL COMPUTERS

In India, the development and design of parallel computers started in the early 80's. The Indian Government established the Centre for Development of Advanced Computing (CDAC) in 1988 with the aim of building high-speed parallel machines. CDAC designed and built a 256 processors computer using INMOS T8000 series processors in 1991. The other groups which developed parallel machines were at Centre for Development of Telematics, Bhabha Atomic Research Centre, Indian Institute for Sciences, Defence Research and Development Organisation. The systems developed by these organisations are said to be the state of the art of parallel computers. It is generally agreed that all the computers built by 2020 will be inherently parallel.

India's Parallel Computer

Next, we enumerate sailent features of various generations of parallel systems developed in India.

Sailent Features of PARAM series:

PARAM 8000 CDAC 1991: 256 Processor parallel computer, INMOS 8000 transputer as processing element. Peak performance of 1 Gigaflop. Application software weak.

PARAM 8600 CDAC 1994: PARAM 8000 enhanced with Intel i860 vector microprocessor. One vector processor for 4 INMOS 8000. Vectorized Fortran. Improved software for numerical applications.

PARAM 9000/SS CDAC 1996 : Used Sunsparc II processors and an interconnection switch made of INMOS transputers.

Salient Features of MARK Series:

Flosolver Mark I NAL 1986: Used 4 Intel 8086 processors with 8087 co-processors. Proof of concept design.

Flosolver Mark II NAL 1988: 16 Intel 80386 processor and 80387 floating-point processor connected to Multibus II backplane bus for interprocessor communication. Used for solving fluid dynamics problems using Fortran.

Flosolver Mark III NAL 1991: 8 Intel i860 vector processors connected using message passing co-processor on a back plane bus. i860 were rated at 80 Mflops peak. Fluid dynamics Codes were optimized for the architecture.

Salient Features of ANUPAM Series:

ANUPAM Model 1 BARC 1993: 8 Intel i860 processors connected to a Multibus II. Eight such clusters connected by using 16-bit SCSI interface. Used Front-end processor to allocate tasks to the parallel computing cluster. One parallel program at a time could be run. Fortran environment.

ANUPAM Model 2 BARC 1997: DEC Alpha processors connected by ATM switch in a cluster. DEC Unix environment. High Performance Fortran compiler to run data parallel programs.

1.11 PARALLEL TERMINOLOGY USED

Some of the more commonly used terms associated with parallel computing are listed below.

Task

A logically discrete section of a computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task

A task, some parts of which can be executed by more than one multiple processor at same point of time (yields correct results)

Serial Execution
Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, even most of the parallel tasks also have some sections of a parallel program that must be executed serially.

Parallel Execution

Execution of sections of a program by more than one processor at the same point of time.

Shared Memory

Refers to the memory component of a computer system in which the memory can accessed directly by any of the processors in the system.

Distributed Memory

Refers to network based memory in which there is no common address space for the various memory modules comprising the memory system of the (networked) system. Generally, a processor or a group of processors have some memory module associated with it, independent of the memory modules associated with other processors or group of processors.

Communications

Parallel tasks typically need to exchange data. There are several ways in which this can be accomplished, such as, through a shared memory bus or over a network. The actual event of data exchange is commonly referred to as communication regardless of the method employed.

Synchronization

The process of the coordination of parallel tasks in real time, very often associated with communications is called synchronisation. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's execution time to increase.

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

Coarse Granularity: relatively large amounts of computational work are done between communication events

Fine Granularity: relatively small amounts of computational work are done between communication events

Observed Speedup
Observed speedup of a code which has been parallelized, is defined as:

$$\frac{\text{wall-clock time of serial}}{\text{wall-clock time of parallel}}$$

Granularity is one of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

Task start-up time
Synchronisations
Data communications
Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.

Massively Parallel System

Refers to a parallel computer system having a large number of processors. The number in 'a large number of' keeps increasing, and, currently it means more than 1000.

Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in (parallel) speedup with the addition of more processors. Factors that contribute to scalability include:

Hardware - particularly memory-cpu bandwidths and network communications:

- *Application algorithm*
- *Parallel overhead related*
- *Characteristics of your specific application and coding*

Check Your Progress 3

1) Explain dataflow computation model.

……
……
……
…….

2) Enumerate applications of parallel processing.

……
……
……
……

1.12 SUMMARY

In this unit, a number of introductory issues and concepts in respect of parallel computing are discussed. First of all, section 1.2 briefly discusses history of parallel computing. Next section discusses two types of parallelism, viz, temporal and data parallelisms. The issues relating to performance evaluation of a parallel system are discussed in section 1.4. Section 1.5 defines a number of new concepts. Next section explains why parallel computation is essential for solving computationally difficult problems. Section 1.7 discusses how parallelism can be achieved at different levels within a program. Dataflow computing is a different paradigm of computing as compared to the most frequently used Von-Neumann-architecture based computing. Dataflow computing allows to exploit easily the inherent parallelism in a problem and its solution. Issues related to Dataflow computing are discussed in section 1.8. Applications of parallel computing are discussed in section 1.9. India's effort at developing parallel computers is briefly discussed in section 1.10. A glossary of parallel computing terms is given in section 1.11.

1.13 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Each process has a life cycle, which consists of creation, execution and termination phases of the process. A process may create several new processes, which in turn may also create still new processes, using system calls. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.

ii) **Allocating an address space:** There are two ways to allocate address space to processes; sharing the address space among the created processes or allocating separate space to each process.

iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.

iv) **Passing the process description to the process scheduler:** The process created is then passed to the process scheduler which allocates the processor to the competing processes.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process states, state transition diagram and scheduling policy.
A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.

- **Terminated:** The process has finished execution.

At any time a process may be in any one of the above said states. As soon as the process is admitted into the job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. When the process is completely executed then it is terminated and we say that it is in the terminated state. However, the process may return to ready state due to some interruption or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown below in the Figure 3:

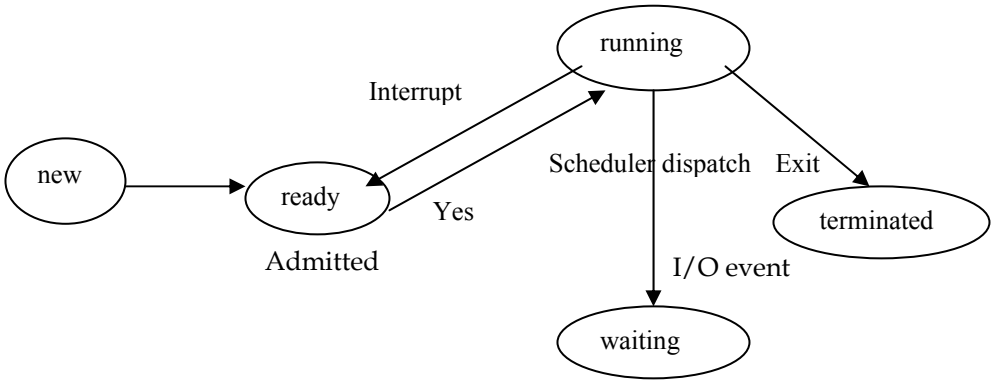


Figure 3: Process state transition diagram

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. One of the well-known policies is First In First Out(FIFO) to select the process to be executed.When the process finishes execution it is terminated by system calls like *abort*.

2) Some of the advantages that threads offer over processes include:

i) It takes less time to create and terminate a new thread than it takes for a process, because the newly created thread uses the current process address space.
ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share the address space, in particular. So, data produced by one thread is immediately available to all the other threads.

3) The words “concurrent” and “parallel” are often used interchangeably, however they are distinct. Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has a dual nature: it is sequential in a small time scale, but simultaneous in large time scale. In our context the processor works as a server and a process or a thread works as a client. For facilitating expression of concurrent programs, number of concurrent languages are available including Ada, concurrent pascal, Modula-2 and concurrent PROLOG.

Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. For facilitating expression of parallel programs, a number of parallel languages are available including: Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

4) Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems parallel parts are relatively small and which means more frequent communication. Fine-grain processings have low computation to communication ratio and require high communication overhead. In coarse grained systems parallel parts are relatively large and which means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amounts of computational work is done. Coarse-grain processings have high computation to communication ratio and imply more opportunity for performance increase.

Check Your Progress 2

1) Parallel computing has the following advantages over sequential computing:

i) Saves time
ii) Solves larger problems.
iii) Large pool of memory.

2) Levels of parallel processing:

We can have parallel processing at four levels.

i) **Instruction Level:** Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe.

ii) **Loop Level:** Here, consecutive loop iterations are candidates for parallel execution. However, data between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

iii) **Procedure Level:** Here parallelism is available in the form of parallel executable procedures. Here the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

iv) **Program Level:** This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by the operating system at this level.

Check Your Progress 3

1) An alternative to the von Neumann model of computation is dataflow computation model. In a dataflow model control is tied to the flow of data. The order of instructions in the program plays no role in the execution order. Computations take place when all the data items needed for initiating execution of an instruction are available. Data is in continuous flow independent of reusable memory cells and its availability initiates execution. Since data may be available for several instructions at the same time, these instructions can be executed in parallel.

The potential for parallel computation, is reflected by the dataflow graph, the nodes of which are the instructions of the program and the edges of which represent data dependency between instructions. The dataflow graph for the instruction z = w × (x+y) is shown below.

.

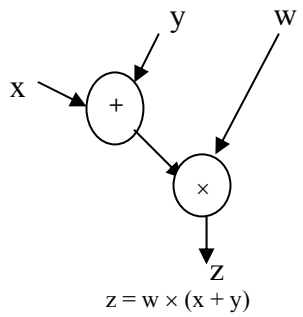


Figure 4: DFG for z = w × (x+y)

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each input edge. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

2) Please refer Section 1.9.

UNIT 2 CLASSIFICATION OF PARALLEL COMPUTERS

Structure **Page Nos.**

2.0 INTRODUCTION

Parallel computers are those that emphasize the parallel processing between the operations in some way. In the previous unit, all the basic terms of parallel processing and computation have been defined. Parallel computers can be characterized based on the data and instruction streams forming various types of computer organisations. They can also be classified based on the computer structure, e.g. multiple processors having separate memory or one shared global memory. Parallel processing levels can also be defined based on the size of instructions in a program called grain size. Thus, parallel computers can be classified based on various criteria. This unit discusses all types of classification of parallel computers based on the above mentioned criteria.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the various criteria on which classification of parallel computers are based;
- discuss the Flynn's classification based on instruction and data streams;
- describe the Structural classification based on different computer organisations;
- explain the Handler's classification based on three distinct levels of computer: Processor control unit (PCU), Arithmetic logic unit (ALU), Bit-level circuit (BLC), and
- describe the sub-tasks or instructions of a program that can be executed in parallel based on the grain size.

2.2 TYPES OF CLASSIFICATION

The following classification of parallel computers have been identified:

1) Classification based on the instruction and data streams
2) Classification based on the structure of computers
3) Classification based on how the memory is accessed
4) Classification based on grain size

All these classification schemes are discussed in subsequent sections.

2.3 FLYNN'S CLASSIFICATION

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of *instruction* and *data* streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

2.3.1 Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See *Figure 1*). The Operand part is divided into two parts: addressing mode and the Operand. The addressing mode specifies the method of determining the addresses of the actual data on which the operation is to be performed and the operand part is used as an argument by the method in determining the actual address.

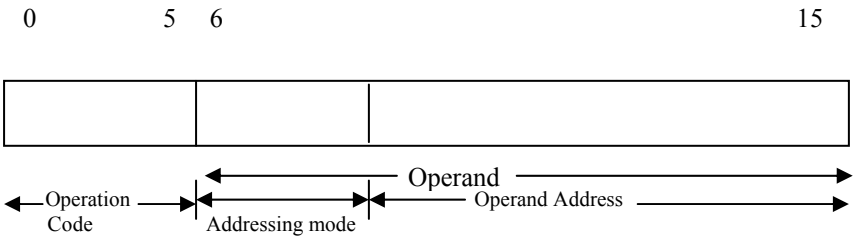


Figure 1: Opcode and Operand

The control unit of the CPU of the computer fetches instructions in the program, one at a time. The fetched Instruction is then decoded by the decoder which is a part of the control unit and the processor executes the decoded instructions. The result of execution is temporarily stored in Memory Buffer Register (MBR) (also called Memory Data Register). The normal execution steps are shown in *Figure 2*.

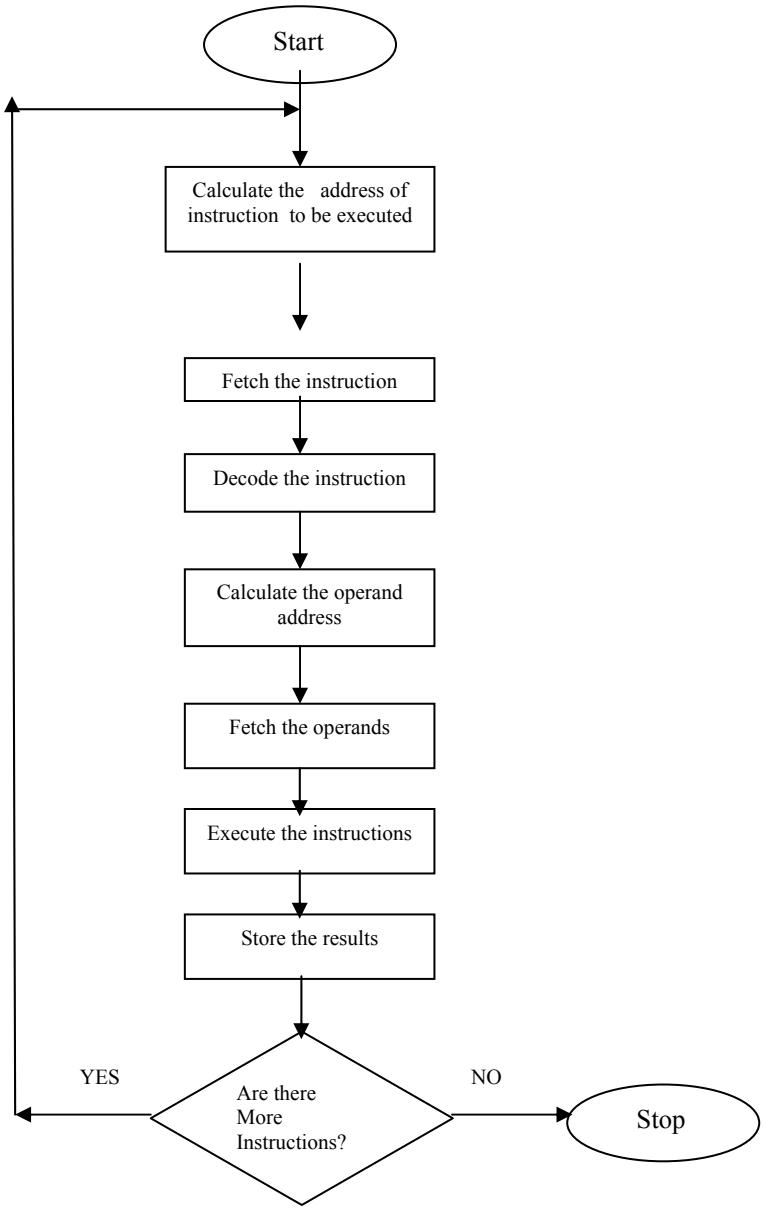


Figure 2: Instruction execution steps

2.3.2 Instruction Stream and Data Stream

The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called ***instruction stream.*** Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called ***data stream***. These two types of streams are shown in *Figure 3*.

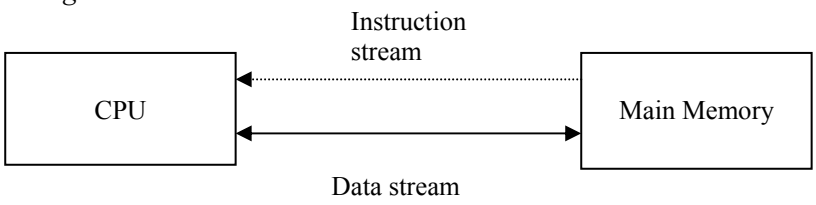


Figure 3: Instruction and data stream

Thus, it can be said that the sequence of instructions executed by CPU forms the Instruction streams and sequence of data (operands) required for execution of instructions form the Data streams.

2.3.3 Flynn's Classification

Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let I_s and D_s are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

1) **Single Instruction and Single Data stream (SISD)**

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in *Figure 4*. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram:

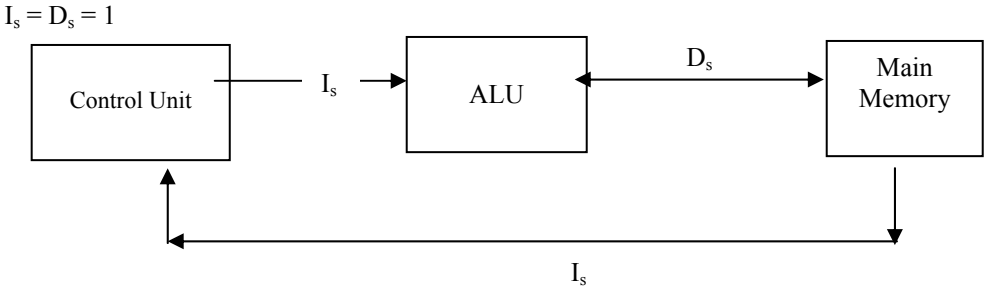


Figure 4: SISD Organisation

Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

2) **Single Instruction and Multiple Data stream (SIMD)**

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a *distributed memory* as shown in *Figure 5*. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as:

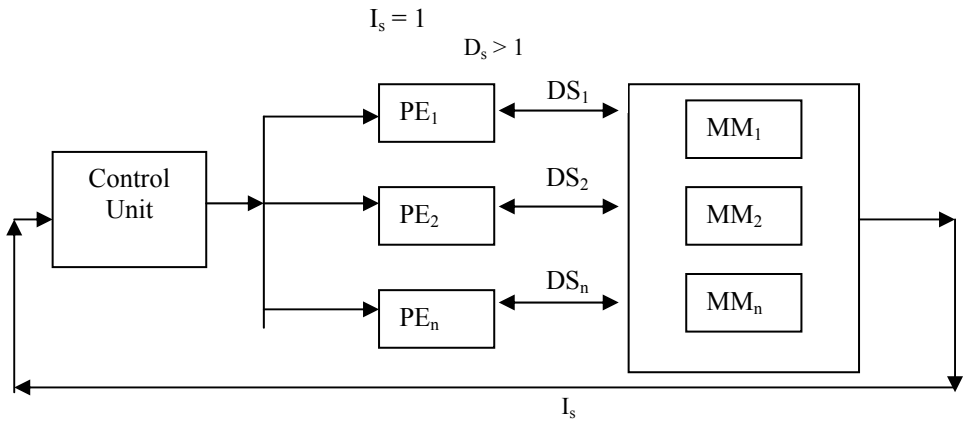


Figure 5: SIMD Organisation

3) Multiple Instruction and Single Data stream (MISD)

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream as shown in *Figure 6*. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.

This type of computer organisation is denoted as:

$I_s > 1$
$D_s = 1$

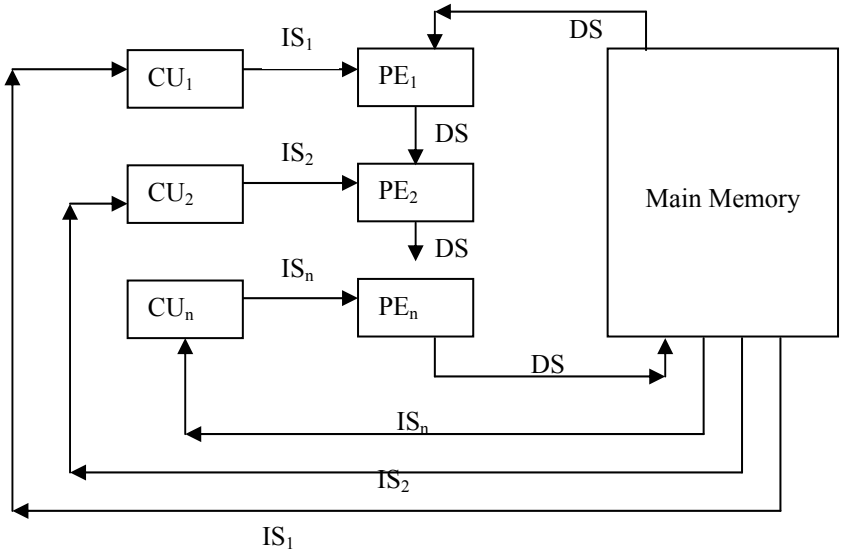


Figure 6: MISD Organisation

This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied. But for the specialized applications, MISD organisation can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N- version programming. All these redundant data

are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

4) **Multiple Instruction and Multiple Data stream (MIMD)**

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams . Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in *Figure 7*. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer. All multiprocessor systems fall under this classification. Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.

This type of computer organisation is denoted as:

$I_s > 1$
$D_s > 1$

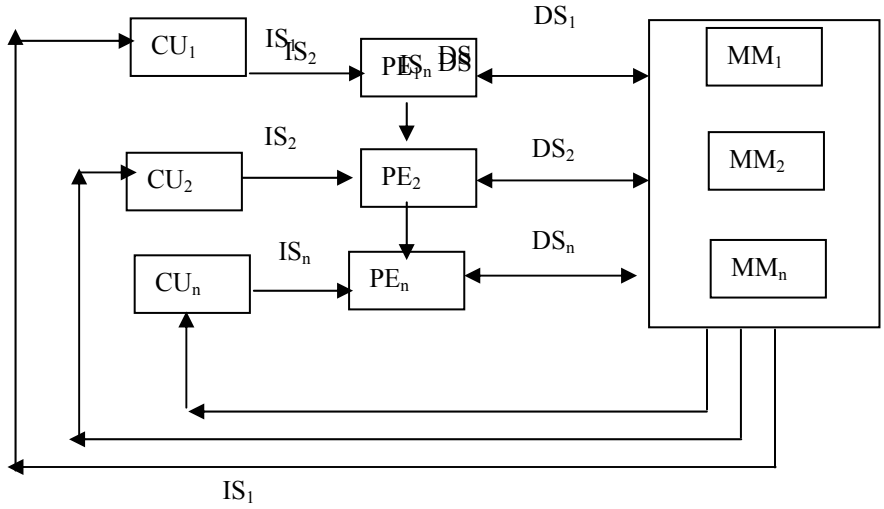


Figure 7: MIMD Organisation

Of the classifications discussed above, MIMD organization is the most popular for a parallel computer. In the real sense, parallel computers execute the instructions in MIMD mode.

Check Your Progress 1

1) What are various criteria for classification of parallel computers?

……
……
……
……

2) Define instruction and data streams.

……
……

……
……

3) State whether True or False for the following:
   a) SISD computers can be characterized as $I_s > 1$ and $D_s > 1$
   b) SIMD computers can be characterized as $I_s > 1$ and $D_s = 1$
   c) MISD computers can be characterized as $I_s = 1$ and $D_s = 1$
   d) MIMD computers can be characterized as $I_s > 1$ and $D_s > 1$

2.4 HANDLER'S CLASSIFICATION

In 1977, Wolfgang Handler proposed an elaborate notation for expressing the pipelining and parallelism of computers. Handler's classification addresses the computer at three distinct levels:

- Processor control unit (PCU),
- Arithmetic logic unit (ALU),
- Bit-level circuit (BLC).

The PCU corresponds to a processor or CPU, the ALU corresponds to a functional unit or a processing element and the BLC corresponds to the logic circuit needed to perform one-bit operations in the ALU.

Handler's classification uses the following three pairs of integers to describe a computer:

Computer = (p * p', a * a', b * b')

Where p = number of PCUs
Where p'= number of PCUs that can be pipelined
Where a = number of ALUs controlled by each PCU
Where a'= number of ALUs that can be pipelined
Where b = number of bits in ALU or processing element (PE) word
Where b'= number of pipeline segments on all ALUs or in a single PE

The following rules and operators are used to show the relationship between various elements of the computer:

- The '*' operator is used to indicate that the units are pipelined or macro-pipelined with a stream of data running through all the units.
- The '+' operator is used to indicate that the units are not pipelined but work on independent streams of data.
- The 'v' operator is used to indicate that the computer hardware can work in one of several modes.
- The '~' symbol is used to indicate a range of values for any one of the parameters.
- Peripheral processors are shown before the main processor using another three pairs of integers. If the value of the second element of any pair is 1, it may omitted for brevity.

Handler's classification is best explained by showing how the rules and operators are used to classify several machines.

The CDC 6600 has a single main processor supported by 10 I/O processors. One control unit coordinates one ALU with a 60-bit word length. The ALU has 10 functional units which can be formed into a pipeline. The 10 peripheral I/O processors may work in parallel with each other and with the CPU. Each I/O processor contains one 12-bit ALU. The description for the 10 I/O processors is:

CDC 6600I/O = (10, 1, 12)

The description for the main processor is:

CDC 6600main = (1, 1 * 10, 60)

The main processor and the I/O processors can be regarded as forming a macro-pipeline so the '*' operator is used to combine the two structures:

CDC 6600 = (I/O processors) * (central processor = (10, 1, 12) * (1, 1 * 10, 60)

Texas Instrument's Advanced Scientific Computer (ASC) has one controller coordinating four arithmetic units. Each arithmetic unit is an eight stage pipeline with 64-bit words. Thus we have:

ASC = (1, 4, 64 * 8)

The Cray-1 is a 64-bit single processor computer whose ALU has twelve functional units, eight of which can be chained together to from a pipeline. Different functional units have from 1 to 14 segments, which can also be pipelined. Handler's description of the Cray-1 is:

Cray-1 = (1, 12 * 8, 64 * (1 ~ 14))

Another sample system is Carnegie-Mellon University's C.mmp multiprocessor. This system was designed to facilitate research into parallel computer architectures and consequently can be extensively reconfigured. The system consists of 16 PDP-11 'minicomputers' (which have a 16-bit word length), interconnected by a crossbar switching network. Normally, the C.mmp operates in MIMD mode for which the description is (16, 1, 16). It can also operate in SIMD mode, where all the processors are coordinated by a single master controller. The SIMD mode description is (1, 16, 16). Finally, the system can be rearranged to operate in MISD mode. Here the processors are arranged in a chain with a single stream of data passing through all of them. The MISD modes description is (1 * 16, 1, 16). The 'v' operator is used to combine descriptions of the same piece of hardware operating in differing modes. Thus, Handler's description for the complete C.mmp is:

C.mmp = (16, 1, 16) v (1, 16, 16) v (1 * 16, 1, 16)

The '*' and '+' operators are used to combine several separate pieces of hardware. The 'v' operator is of a different form to the other two in that it is used to combine the different operating modes of a single piece of hardware.

While Flynn's classification is easy to use, Handler's classification is cumbersome. The direct use of numbers in the nomenclature of Handler's classification's makes it much more abstract and hence difficult. Handler's classification is highly geared towards the description of pipelines and chains. While it is well able to describe the parallelism in a single processor, the variety of parallelism in multiprocessor computers is not addressed well.

2.5 STRUCTURAL CLASSIFICATION

Flynn's classification discusses the behavioural concept and does not take into consideration the computer's structure. Parallel computers can be classified based on their structure also, which is discussed below and shown in *Figure 8*.

As we have seen, a parallel computer (MIMD) can be characterised as a set of multiple processors and shared memory or memory modules communicating via an interconnection network. When multiprocessors communicate through the global shared memory modules then this organisation is called ***Shared memory computer*** or ***Tightly***

coupled systems as shown in *Figure 9.* Similarly when every processor in a multiprocessor system, has its own local memory and the processors communicate via messages transmitted between their local memories, then this organisation is called ***Distributed memory computer*** or ***Loosely coupled system*** as shown in *Figure 10*. *Figures 9* and *10* show the simplified diagrams of both organisations.

The processors and memory in both organisations are interconnected via an interconnection network. This interconnection network may be in different forms like crossbar switch, multistage network, etc. which will be discussed in the next unit.

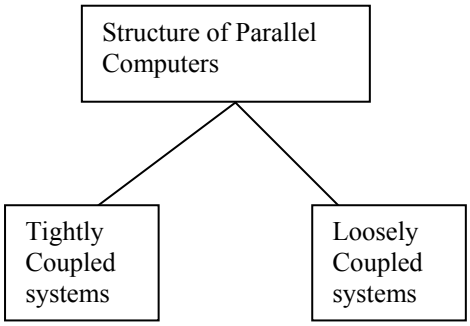


Figure 8: Structural classification

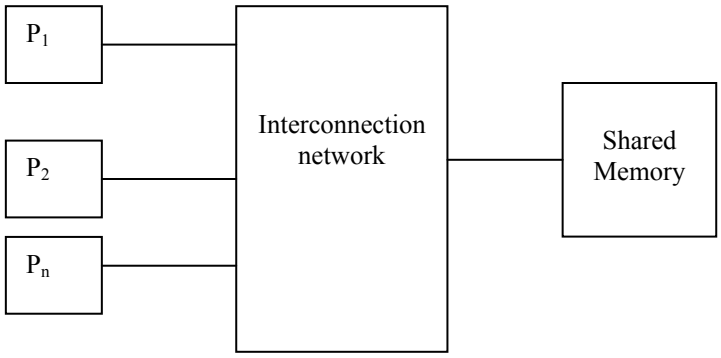


Figure 9: Tightly coupled system

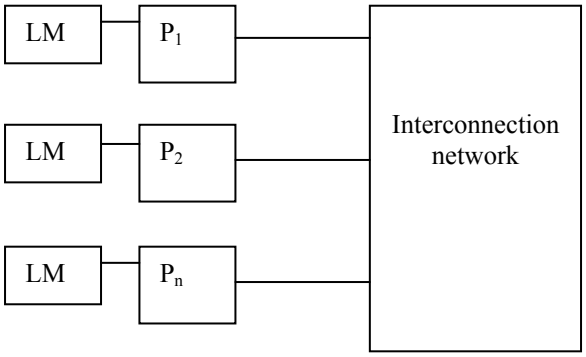


Figure 10 Loosely coupled system

2.5.1 Shared Memory System / Tightly Coupled System

Shared memory multiprocessors have the following characteristics:

- Every processor communicates through a shared global memory.

- For high speed real time processing, these systems are preferable as their throughput is high as compared to loosely coupled systems.

In tightly coupled system organization, multiple processors share a global main memory, which may have many modules as shown in detailed *Figure 11*. The processors have also access to I/O devices. The inter- communication between processors, memory, and other devices are implemented through various interconnection networks, which are discussed below.

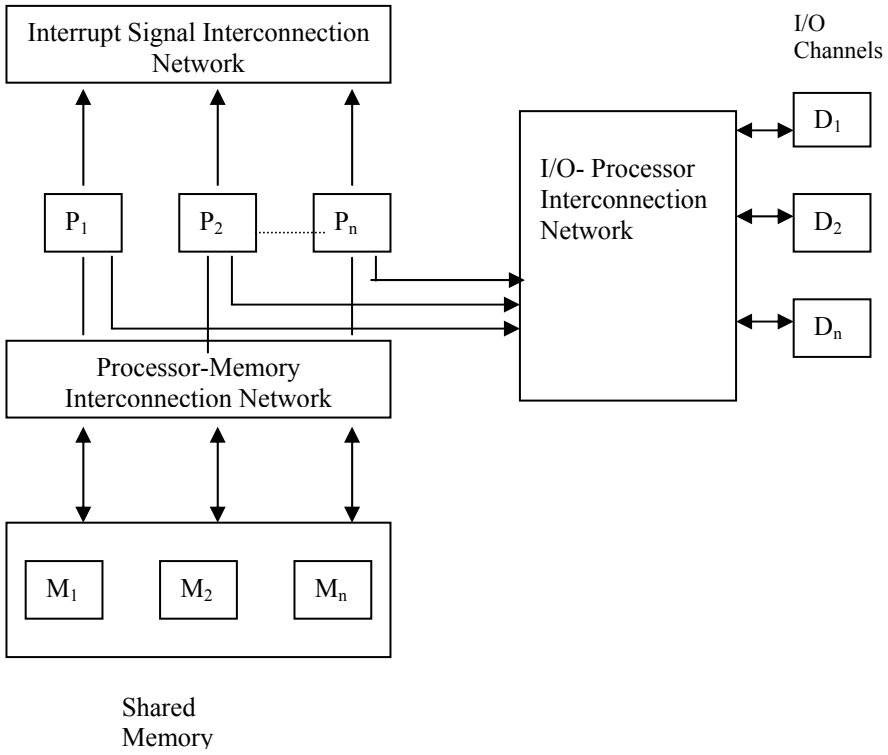


Figure 11: Tightly coupled system organization

i) **Processor-Memory Interconnection Network (PMIN)**

This is a switch that connects various processors to different memory modules. Connecting every processor to every memory module in a single stage while the crossbar switch may become complex. Therefore, multistage network can be adopted. There can be a conflict among processors such that they attempt to access the same memory modules. This conflict is also resolved by PMIN.

ii) **Input-Output-Processor Interconnection Network (IOPIN)**

This interconnection network is used for communication between processors and I/O channels. All processors communicate with an I/O channel to interact with an I/O device with the prior permission of IOPIN.

iii) **Interrupt Signal Interconnection Network (ISIN)**

When a processor wants to send an interruption to another processor, then this interrupt first goes to ISIN, through which it is passed to the destination processor. In this way, synchronisation between processor is implemented by ISIN. Moreover, in case of failure of one processor, ISIN can broadcast the message to other processors about its failure.

Since, every reference to the memory in tightly coupled systems is via interconnection network, there is a delay in executing the instructions. To reduce this delay, every

processor may use cache memory for the frequent references made by the processor as shown in *Figure 12*.

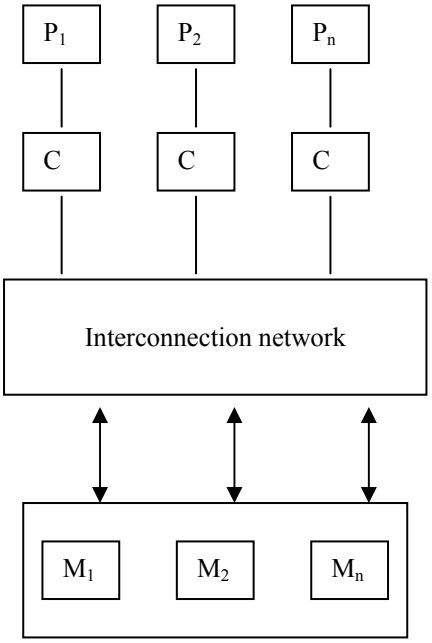


Figure 12: Tightly coupled systems with cache memory

The shared memory multiprocessor systems can further be divided into three modes which are based on the manner in which shared memory is accessed. These modes are shown in *Figure 13* and are discussed below.

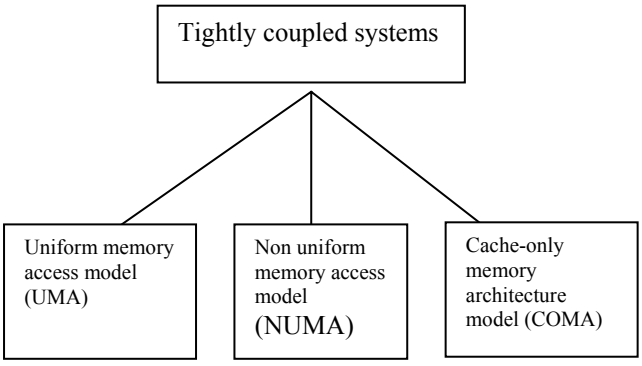


Figure 13: Modes of Tightly coupled systems

2.5.1.1 Uniform Memory Access Model (UMA)

In this model, main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory. This model is used for time-sharing applications in a multi user environment.

2.5.1.2 Non-Uniform Memory Access Model (NUMA)

In shared memory multiprocessor systems, local memories can be connected with every processor. The collection of all local memories form the global memory being shared. In this way, global memory is distributed to all the processors. In this case, the access to a local memory is uniform for its corresponding processor as it is attached to the local memory. But if one reference is to the local memory of some other remote processor, then

the access is not uniform. It depends on the location of the memory. Thus, all memory words are not accessed uniformly.

2.5.1.3 Cache-Only Memory Access Model (COMA)

As we have discussed earlier, shared memory multiprocessor systems may use cache memories with every processor for reducing the execution time of an instruction. Thus in NUMA model, if we use cache memories instead of local memories, then it becomes COMA model. The collection of cache memories form a global memory space. The remote cache access is also non-uniform in this model.

2.5.2 Loosely Coupled Systems

These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions. Therefore, to alleviate this problem, each processor in loosely coupled systems is having a large local memory (LM), which is not shared by any other processor. Thus, such systems have multiple processors with their own local memory and a set of I/O devices. This set of processor, memory and I/O devices makes a computer system. Therefore, these systems are also called multi-computer systems. These computer systems are connected together via message passing interconnection network through which processes communicate by passing messages to one another. Since every computer system or node in multicomputer systems has a separate memory, they are called distributed multicomputer systems. These are also called loosely coupled systems, meaning that nodes have little coupling between them as shown in *Figure 14*.

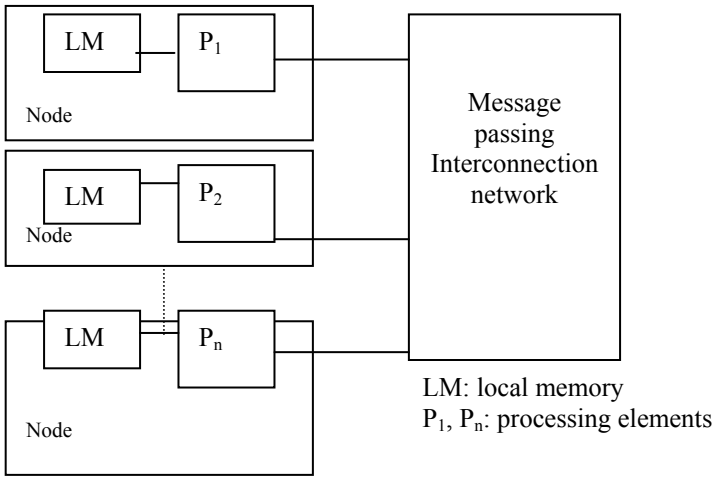


Figure 14: Loosely coupled system organisation

Since local memories are accessible to the attached processor only, no processor can access remote memory. Therefore, these systems are also known as no-remote memory access (NORMA) systems. Message passing interconnection network provides connection to every node and inter-node communication with message depends on the type of interconnection network. For example, interconnection network for a non-hierarchical system can be shared bus.

Check Your Progress 2

1) What are the various rules and operators used in Handler's classification for various machine types?

……..
……
……
..……

2) What is the base for structural classification of parallel computers?
……..
……..
……..
……..

3) Define loosely coupled systems and tightly coupled systems.
…….
……
..……..
…..……..

4) Differentiate between UMA, NUMA and COMA.
……..
……..
……..
……..

2.6 CLASSIFICATION BASED ON GRAIN SIZE

This classification is based on recognizing the parallelism in a program to be executed on a multiprocessor system. The idea is to identify the sub-tasks or instructions in a program that can be executed in parallel. For example, there are 3 statements in a program and statements S1 and S2 can be exchanged. That means, these are not sequential as shown in *Figure 15*. Then S1 and S2 can be executed in parallel.

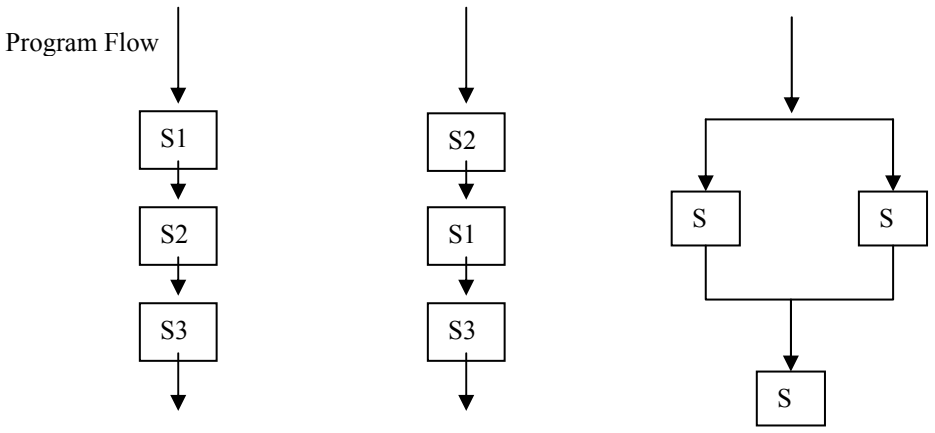


Figure 15: Parallel execution for S1 and S2

But it is not sufficient to check for the parallelism between statements or processes in a program. The decision of parallelism also depends on the following factors:

- Number and types of processors available, i.e., architectural features of host computer
- Memory organisation
- Dependency of data, control and resources

2.6.1 Parallelism Conditions

As discussed above, parallel computing requires that the segments to be executed in parallel must be independent of each other. So, before executing parallelism, all the conditions of parallelism between the segments must be analyzed. In this section, we discuss three types of dependency conditions between the segments (shown in *Figure 16*).

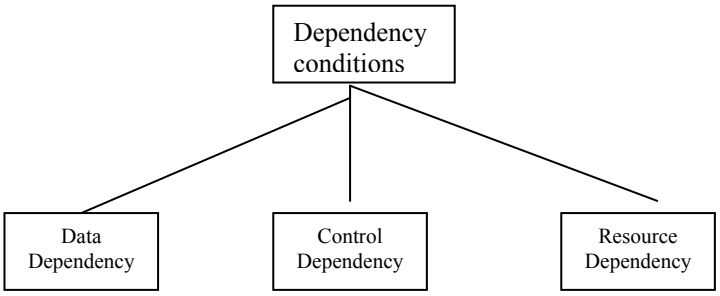


Figure 16: Dependency relations among the segments for parallelism

Data Dependency: It refers to the situation in which two or more instructions share same data. The instructions in a program can be arranged based on the relationship of data dependency; this means how two instructions or segments are data dependent on each other. The following types of data dependencies are recognised:

i) **Flow Dependence :** If instruction I_2 follows I_1 and output of I_1 becomes input of I_2, then I_2 is said to be flow dependent on I_1.
ii) **Antidependence :** When instruction I_2 follows I_1 such that output of I_2 overlaps with the input of I_1 on the same data.
iii) **Output dependence :** When output of the two instructions I_1 and I_2 overlap on the same data, the instructions are said to be output dependent.
iv) **I/O dependence :** When read and write operations by two instructions are invoked on the same file, it is a situation of I/O dependence.

Consider the following program instructions:

I_1: a = b
I_2: c = a + d
I_3: a = c

In this program segment instructions I_1 and I_2 are Flow dependent because variable a is generated by I_1 as output and used by I_2 as input. Instructions I_2 and I_3 are Antidependent because variable a is generated by I_3 but used by I_2 and in sequence I_2 comes first. I_3 is flow dependent on I_2 because of variable c. Instructions I_3 and I_1 are Output dependent because variable a is generated by both instructions.

Control Dependence: Instructions or segments in a program may contain control structures. Therefore, dependency among the statements can be in control structures also. But the order of execution in control structures is not known before the run time. Thus, control structures dependency among the instructions must be analyzed carefully. For example, the successive iterations in the following control structure are dependent on one another.

```
For ( i= 1; I<= n ; i++)
{
        if (x[i - 1] == 0)
                x[i] =0
        else
                x[i] = 1;
}
```

Resource Dependence : The parallelism between the instructions may also be affected due to the shared resources. If two instructions are using the same shared resource then it is a resource dependency condition. For example, floating point units or registers are shared, and this is known as *ALU dependency.* When memory is being shared, then it is called S*torage dependency.*

2.6.2 Bernstein Conditions for Detection of Parallelism

For execution of instructions or block of instructions in parallel, it should be ensured that the instructions are independent of each other. These instructions can be data dependent / control dependent / resource dependent on each other. Here we consider only data dependency among the statements for taking decisions of parallel execution. **A.J. Bernstein** has elaborated the work of data dependency and derived some conditions based on which we can decide the parallelism of instructions or processes.

Bernstein conditions are based on the following two sets of variables:

i) The Read set or input set R_I that consists of memory locations read by the statement of instruction I_1.
ii) The Write set or output set W_I that consists of memory locations written into by instruction I_1.

The sets R_I and W_I are not disjoint as the same locations are used for reading and writing by S_I.

The following are Bernstein Parallelism conditions which are used to determine whether statements are parallel or not:

1) Locations in R_1 from which S_1 reads and the locations W_2 onto which S_2 writes must be mutually exclusive. That means S_1 does not read from any memory location onto which S_2 writes. It can be denoted as:
$R_1 \cap W_2 = \phi$

2) Similarly, locations in R_2 from which S_2 reads and the locations W_1 onto which S_1 writes must be mutually exclusive. That means S_2 does not read from any memory location onto which S_1 writes. It can be denoted as: $R_2 \cap W_1 = \phi$

3) The memory locations W_1 and W_2 onto which S_1 and S_2 write, should not be read by S_1 and S_2. That means R_1 and R_2 should be independent of W_1 and W_2. It can be denoted as : $W_1 \cap W_2 = \phi$

To show the operation of Bernstein's conditions, consider the following instructions of sequential program:

I1 : x = (a + b) / (a * b)
I2 : y = (b + c) * d
I3 : z = x^2 + (a * e)

Now, the read set and write set of I1, I2 and I3 are as follows:

$R_1 = \{a,b\}$ $W_1 = \{x\}$
$R_2 = \{b,c,d\}$ $W_2 = \{y\}$
$R_3 = \{x,a,e\}$ $W_3 = \{z\}$

Now let us find out whether I_1 and I_2 are parallel or not

$R_1 \cap W_2 = \phi$
$R_2 \cap W_1 = \phi$
$W_1 \cap W_2 = \phi$

That means I_1 and I_2 are independent of each other.
Similarly for $I_1 \parallel I_3$,

$R_1 \cap W_3 = \phi$
$R_3 \cap W_1 \neq \phi$
$W_1 \cap W_3 = \phi$

Hence I_1 and I_3 are not independent of each other.
For $I_2 \parallel I_3$,

$R_2 \cap W_3 = \phi$
$R_3 \cap W_2 = \phi$

$W_3 \cap W_2 = \phi$

Hence, I_2 and I_3 are independent of each other.
Thus, I_1 and I_2, I_2 and I_3 are parallelizable but I_1 and I_3 are not.

2.6.3 Parallelism based on Grain size

Grain size: Grain size or Granularity is a measure which determines how much computation is involved in a process. Grain size is determined by counting the number of instructions in a program segment. The following types of grain sizes have been identified (shown in *Figure 17*):

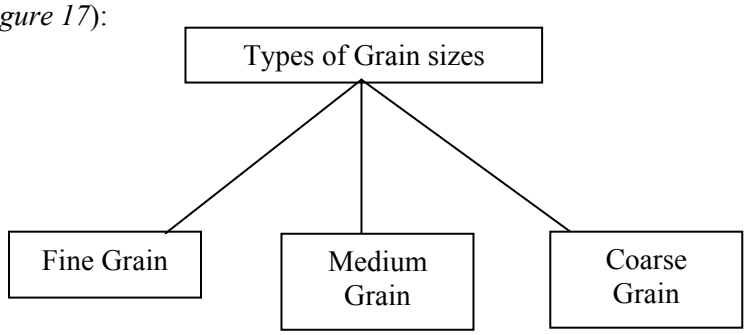


Figure 17: Types of Grain sizes

1) **Fine Grain:** This type contains approximately less than 20 instructions.
2) **Medium Grain:** This type contains approximately less than 500 instructions.
3) **Coarse Grain:** This type contains approximately greater than or equal to one thousand instructions.

Based on these grain sizes, parallelism can be classified at various levels in a program. These parallelism levels form a hierarchy according to which, lower the level, the finer is the granularity of the process. The degree of parallelism decreases with increase in level. Every level according to a grain size demands communication and scheduling overhead. Following are the parallelism levels (shown in *Figure 18*):

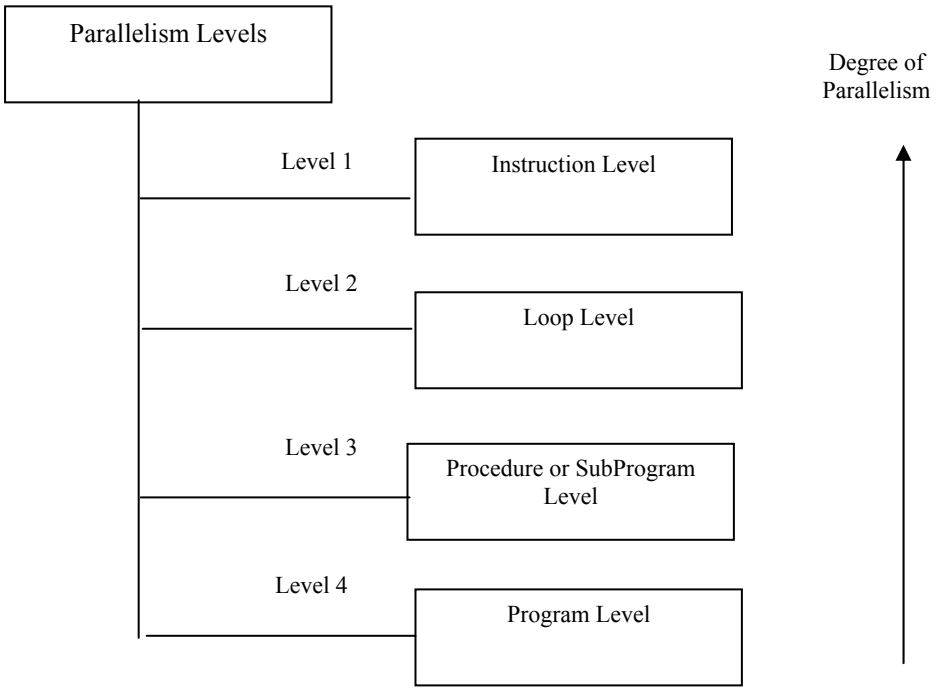


Figure 18: Parallelism Levels

1) **Instruction level:** This is the lowest level and the degree of parallelism is highest at this level. The fine grain size is used at instruction or statement level as only few instructions form the grain size here. The fine grain size may vary according to the type of the program. For example, for scientific applications, the instruction level grain size may be higher. As the higher degree of parallelism can be achieved at this level, the overhead for a programmer will be more.
2) **Loop Level :** This is another level of parallelism where iterative loop instructions can be parallelized. Fine grain size is used at this level also. Simple loops in a program are easy to parallelize whereas the recursive loops are difficult. This type of parallelism can be achieved through the compilers.

3) **Procedure or SubProgram Level:** This level consists of procedures, subroutines or subprograms. Medium grain size is used at this level containing some thousands of instructions in a procedure. Multiprogramming is implemented at this level. Parallelism at this level has been exploited by programmers but not through compilers. Parallelism through compilers has not been achieved at the medium and coarse grain size.

4) **Program Level:** It is the last level consisting of independent programs for parallelism. Coarse grain size is used at this level containing tens of thousands of instructions. Time sharing is achieved at this level of parallelism. Parallelism at this level has been exploited through the operating system.

The relation between grain sizes and parallelism levels has been shown in *Table 1*.

Table 1: Relation between grain sizes and parallelism

Grain Size	Parallelism Level
Fine Grain	Instruction or Loop Level
Medium Grain	Procedure or SubProgram Level
Coarse Grain	Program Level

Coarse grain parallelism is traditionally implemented in tightly coupled or shared memory multiprocessors like the Cray Y-MP. Loosely coupled systems are used to execute medium grain program segments. Fine grain parallelism has been observed in SIMD organization of computers.

Check Your Progress 3

1) Determine the dependency relations among the following instructions:
   I1: a = b+c;
   I2: b = a+d;
   I3: e = a/ f;
   ……
   ……
   ……
   ……

2) Use Bernstein's conditions for determining the maximum parallelism between the instructions in the following segment:
   S1: X = Y + Z
   S2: Z = U + V
   S3: R = S + V
   S4: Z = X + R
   S5: Q = M + Z

……
……
……
……

3) Discuss instruction level parallelism.

……
……
……
……

2.7 SUMMARY

In section 2.3, we discussed Flynn's Classification of computers. This classification scheme was suggested by Michael Flynn in 1972 and is based on the concepts of data stream and instruction stream. Next, we discuss Handler's classification scheme in section 2.4. This classification scheme, suggested by Wolfgang Handler in 1977, addresses the computers at the following three distinct levels:

- Processor Control Unit (PCU)
- Arithmetic Logic Unit (ALU)
- Bit-Level Circuit (BLC)

In section 2.5, in context of structural classification of computers, a number of new concepts are introduced and discussed. The concepts discussed include: Tightly Coupled (or shared memory) systems, loosely coupled (or distributed memory) systems. In the case of distributed memory systems, different types of Processor Interconnection Networks (PIN) are discussed. Another classification scheme based on the concept of grain size is discussed in section 2.6.

2.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) The following criteria have been identified for classifying parallel computers:
   - Classification based on instruction and data streams
   - Classification based on the structure of computers
   - Classification based on how the memory is accessed
   - Classification based on grain size
2) The flow of instructions from the main memory to the CPU is called instruction stream and a flow of operands between processor and memory bi-directionally is known as data stream.
3) a) F
   b) F
   c) F
   d) T

Check Your Progress 2

1) The following rules and operators are used to show the relationship between various elements of the computer:
   - The '*' operator is used to indicate that the units are pipelined or macro-pipelined with a stream of data running through all the units.

- The '+' operator is used to indicate that the units are not pipelined but work on independent streams of data.
- The 'v' operator is used to indicate that the computer hardware can work in one of several modes.
- The '~' symbol is used to indicate a range of values for any one of the parameters.
- Peripheral processors are shown before the main processor using another three pairs of integers. If the value of the second element of any pair is 1, it may be omitted for brevity.

1) The base for structural classification is multiple processors with memory being globally shared between processors or all the processors have their local copy of the memory.

1) When multiprocessors communicate through the global shared memory modules then this organization
is called shared memory computer or tightly coupled systems . When every processor in a multiprocessor system, has its own local memory and the processors communicate via messages transmitted between their local memories, then this organization is called distributed memory computer or loosely coupled system.

1) In UMA, each processor has equal access time to shared memory. In NUMA, local memories are connected with every processor and one reference to a local memory of the remote processor is not uniform. In COMA, all local memories of NUMA are replaced with cache memories.

Check Your Progress 3

1) Instructions I1 and I2 are both flow dependent and antidependent both. Instruction I2 and I3 are output dependent and instructions I1 and I3 are independent.

2) $R_1 = \{Y,Z\}$ $W_1 = \{X\}$
$R_2 = \{U,V\}$ $W_2 = \{Z\}$
$R_3 = \{S,V\}$ $W_3 = \{R\}$
$R_4 = \{X,R\}$ $W_4 = \{Z\}$
$R_5 = \{M,Z\}$ $W_5 = \{Q\}$

Thus, S1, S3 and S5 and S2 & S4 are parallelizable.

3) This is the lowest level and the degree of parallelism is highest at this level. The fine grain size is used at instruction or statement level as only few instructions form the grain size here. The fine grain size may vary according to the type of the program. For example, for scientific applications, the instruction level grain size may be higher. The loops As the higher degree of parallelism can be achieved at this level, the overhead for a programmer will be more.

UNIT 3 INTERCONNECTION NETWORK

Structure **Page Nos.**

3.0 INTRODUCTION

This unit discusses the properties and types of interconnection networks. In multiprocessor systems, there are multiple processing elements, multiple I/O modules, and multiple memory modules. Each processor can access any of the memory modules and any of the I/O units. The connectivity between these is performed by interconnection networks.

Thus, an interconnection network is used for exchanging data between two processors in a multistage network. Memory bottleneck is a basic shortcoming of Von Newman architecture. In case of multiprocessor systems, the performance will be severely affected in case the data exchange between processors is delayed. The multiprocessor system has one global shared memory and each processor has a small local memory. The processors can access data from memory associated with another processor or from shared memory using an interconnection network. Thus, interconnection networks play a central role in determining the overall performance of the multiprocessor systems. The interconnection networks are like customary network systems consisting of nodes and edges. The nodes are switches having few input and few output (say n input and m output) lines. Depending upon the switch connection, the data is forwarded from input lines to output lines. The interconnection network is placed between various devices in the multiprocessor network.

The architecture of a general multiprocessor is shown in *Figure 1*. In the multiprocessor systems, these are multiple processor modules (each processor module consists of a processing element, small sized local memory and cache memory), shared global memory and shared peripheral devices.

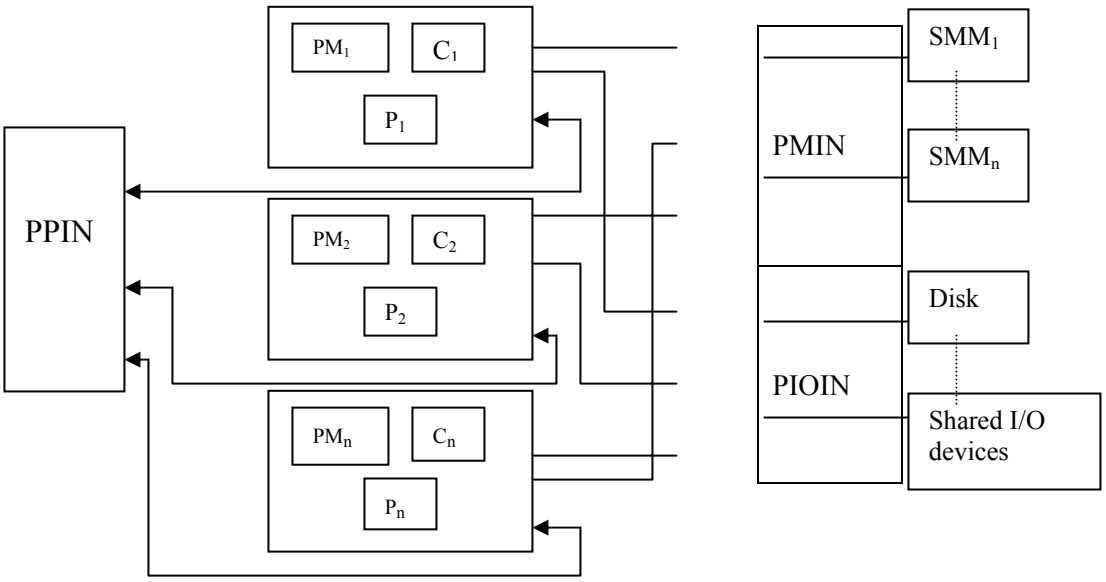


Figure 1: General Multi-Processor

PMIN = Processor to Memory Interconnection Network
PIOIN= Processor to I/O Interconnection Network
PPIN = Processor to Processor Interconnection Network
PM = Processor Module

Module communicates with other modules shared memory and peripheral devices using interconnection networks.

3.1 OBJECTIVES

After studying this unit, students will be able to understand

- discuss the meaning and needs of interconnection network;
- describe the role of interconnection network in a multiprocessor system;
- enumerate the types of interconnection network;
- explain the concept of permutation network;
- discuss the various interconnection networks, and
- describe how matrix multiplication can be carried out on an interconnection network.

3.2 NETWORK PROPERTIES

The following properties are associated with interconnection networks.

1) *Topology:* It indicates how the nodes a network are organised. Various topologies are discussed in Section 3.5.

2) *Network Diameter*: It is the minimum distance between the farthest nodes in a network. The distance is measured in terms of number of distinct hops between any two nodes.

3) *Node degree*: Number of edges connected with a node is called node degree. If the edge carries data from the node, it is called out degree and if this carries data into the node it is called in degree.

4) *Bisection Bandwidth:* Number of edges required to be cut to divide a network into two halves is called bisection bandwidth.

5) *Latency:* It is the delay in transferring the message between two nodes.

6) *Network throughput:* It is an indicative measure of the message carrying capacity of a network. It is defined as the total number of messages the network can transfer per unit time. To estimate the throughput, the capacity of the network and the messages number of actually carried by the network are calculated. Practically the throughput is only a fraction of its capacity.
In interconnection network the traffic flow between nodes may be nonuniform and it may be possible that a certain pair of nodes handles a disproportionately large amount of traffic. These are called "hot spot." The hot spot can behave as a bottleneck and can degrade the performance of the entire network.

7) *Data Routing Functions:* The data routing functions are the functions which when executed establishe the path between the source and the destination. In dynamic interconnection networks there can be various interconnection patterns that can be generated from a single network. This is done by executing various data routing functions. Thus data routing operations are used for routing the data between various processors. The data routing network can be static or dynamic static network

8) *Hardware Cost:* It refers to the cost involved in the implementation of an interconnection network. It includes the cost of switches, arbiter unit, connectors, arbitration unit, and interface logic.

9) *Blocking and Non-Blocking network:* In non-blocking networks the route from any free input node to any free output node can always be provided. Crossbar is an example of non-blocking network. In a blocking network simultaneous route establishment between a pair of nodes may not be possible. There may be situations where blocking can occur. Blocking refers to the situation where one switch is required to establish more than one connection simultaneously and end-to-end path cannot be established even if the input nodes and output nodes are free. The example of this is a blocking multistage network.

10) *Static and Dynamic Interconnection Network:* In a static network the connection between input and output nodes is fixed and cannot be changed. Static interconnection network cannot be reconfigured. The examples of this type of network are linear array, ring, chordal ring, tree, star, fat tree, mesh, tours, systolic arrays, and hypercube. This type of interconnection networks are more suitable for building computers where the communication pattern is more or less fixed, and can be implemented with static connections. In dynamic network the interconnection pattern between inputs and outputs can be changed. The interconnection pattern can be reconfigured according to the program demands. Here, instead of fixed connections, the switches or arbiters are used. Examples of such networks are buses, crossbar switches, and multistage networks. The dynamic networks are normally used in shared memory(SM) multiprocessors.

11) *Dimensionality of Interconnection Network:* Dimensionality indicates the arrangement of nodes or processing elements in an interconnection network. In single dimensional or linear network, nodes are connected in a linear fashion; in two dimensional network the processing elements (PE's) are arranged in a grid and in cube network they are arranged in a three dimensional network.

12) *Broadcast and Multicast :*In the broadcast interconnection network, at one time one node transmits the data and all other nodes receive that data. Broadcast is one to all mapping. It is the implementation achieved by SIMD computer systems. Message passing multi-computers also have broadcast networks. In multicast network many nodes are simultaneously allowed to transmit the data and multiple nodes receive the data.

3.3 DESIGN ISSUES OF INTERCONNECTION NETWORK

The following are the issues, which should be considered while designing an interconnection network.

1) *Dimension and size of network:* It should be decided how many PE's are there in the network and what the dimensionality of the network is i.e. with how many neighburs, each processor is connected.
2) *Symmetry of the network:* It is important to consider whether the network is symmetric or not i.e., whether all processors are connected with same number of processing elements, or the processing elements of corners or edges have different number of adjacent elements.

3) *What is data communication strategy?* Whether all processors are communicating with each other in one time unit synchronously or asynchronously on demand basis.

4) *Message Size:* What is message size? How much data a processor can send in one time unit.

5) *Start up time:* What is the time required to initiate the communication process.

6) *Data transfer time:* How long does it take for a message to reach to another processor. Whether this time is a function of link distance between two processors or it depends upon the number of nodes coming in between.

7) *The interconnection network is static or dynamic:* That means whether the configuration of interconnection network is governed by algorithm or the algorithm allows flexibility in choosing the path.

Check Your Progress 1

1) Define the following terms related with interconnection networks.
   i) Node degrees
   ii) Dynamic connection network
   iii) Network diameter

……
……
……

2) What is the significance of a bisection bandwidth?

……

……

……

3.4 VARIOUS INTERCONNECTION NETWORKS

In this section, we will discuss some simple and popularly used interconnection networks

1) Fully connected: This is the most powerful interconnection topology. In this each node is directly connected to all other nodes. The shortcoming of this network is that it requires too many connections.

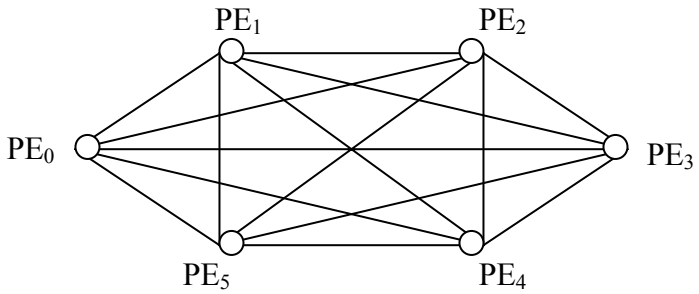


Figure 2: Fully connected interconnection topology

2) Cross Bar: The crossbar network is the simplest interconnection network. It has a two dimensional grid of switches. It is a non-blocking network and provides connectivity between inputs and outputs and it is possible to join any of the inputs to any output.

An N * M crossbar network is shown in the following *Figure 3 (a)* and switch connections are shown in *Figure 3 (b)*.

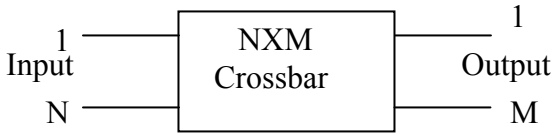


Figure: 3(a)

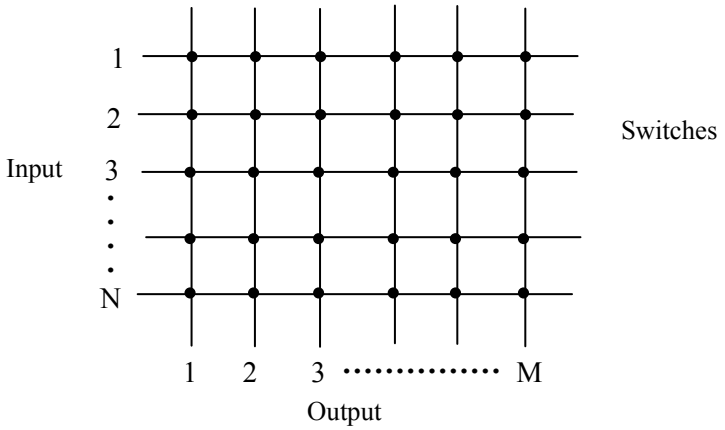


Figure: 3(b)
Figure 3: Crossbar Network

A switch positioned at a cross point of a particular row and particular column. connects that particular row (input) to column (output).

The hardware cost of N*N crossbar switch is proportional to N^2. It creates delay equivalent to one switching operation and the routing control mechanism is easy. The crossbar network requires N^2 switches for N input and N output network.

3) *Linear Array:* This is a most fundamental interconnection pattern. In this processors are connected in a linear one-dimensional array. The first and last processors are connected with one adjacent processor and the middle processing elements are connected with two adjacent processors. It is a one-dimensional interconnection network.

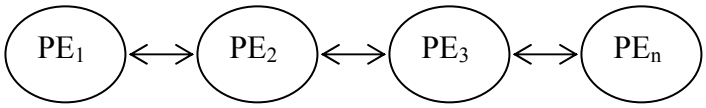


Figure 4: Linear Array

4) *Mesh:* It is a two dimensional network. In this all processing elements are arranged in a two dimensional grid. The processor in rows *i* and column *j* are denoted by PE_i.

The processors on the corner can communicate to two nearest neighbors i.e. PE_{00} can communicate with PE_{01} and PE_{10}. The processor on the boundary can communicate to 3 adjacent processing elements i.e. PE_{01} can communicate with $PE_{00,}$ PE_{02} and PE_{11} and internally placed processors can communicate with 4 adjacent processors i.e. PE_{11} can communicate with $PE_{01,}$ $PE_{10,}$ $PE_{12,}$ and PE_{21}

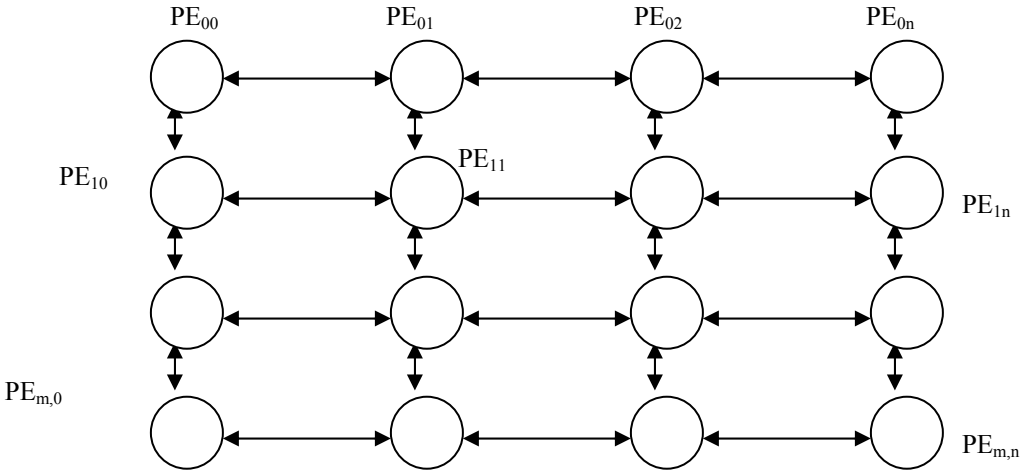


Figure 5: Mesh Network

5) *Ring:* This is a simple linear array where the end nodes are connected. It is equivalent to a mesh with wrap around connections. The data transfer in a ring is normally one direction. Thus, one drawback to this network is that some data transfer may require N/2 links to be traveled (like nodes 2 & 1) where N is the total number of nodes.

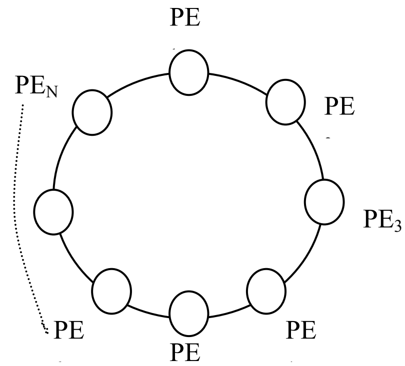


Figure 6: Ring network

6) Torus: The mesh network with wrap around connections is called Tours Network.

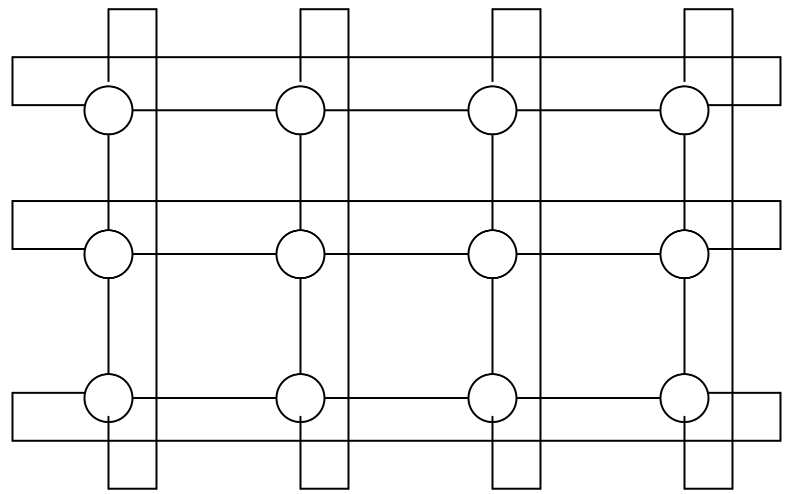


Figure 7: Torus network

7) Tree interconnection network: In the tree interconnection network, processors are arranged in a complete binary tree pattern.

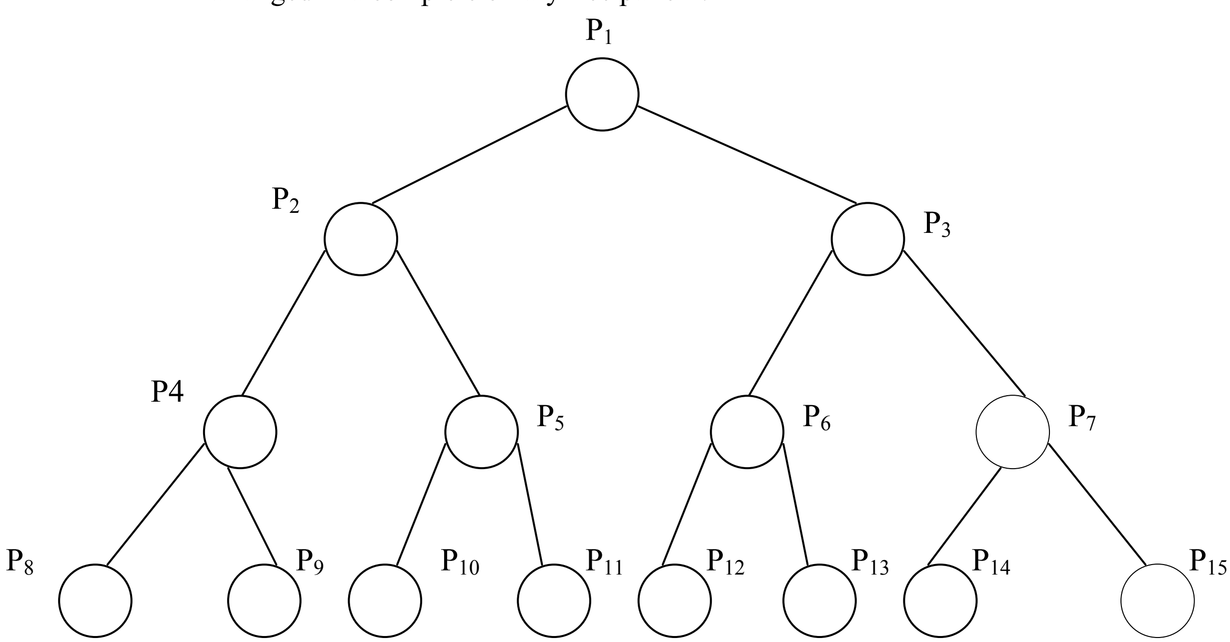


Figure 8:Tree interconnection network

8) Fat tree: It is a modified version of the tree network. In this network the bandwidth of edge (or the connecting wire between nodes) increases towards the root. It is a more realistic simulation of the normal tree where branches get thicker towards root. It is the more popular as compared to tree structure, because practically the more traffic occurs towards the root as compared to leaves, thus if bandwidth remains the same the root will be a bottleneck causing more delay. In a tree this problem is avoided because of higher bandwidth.

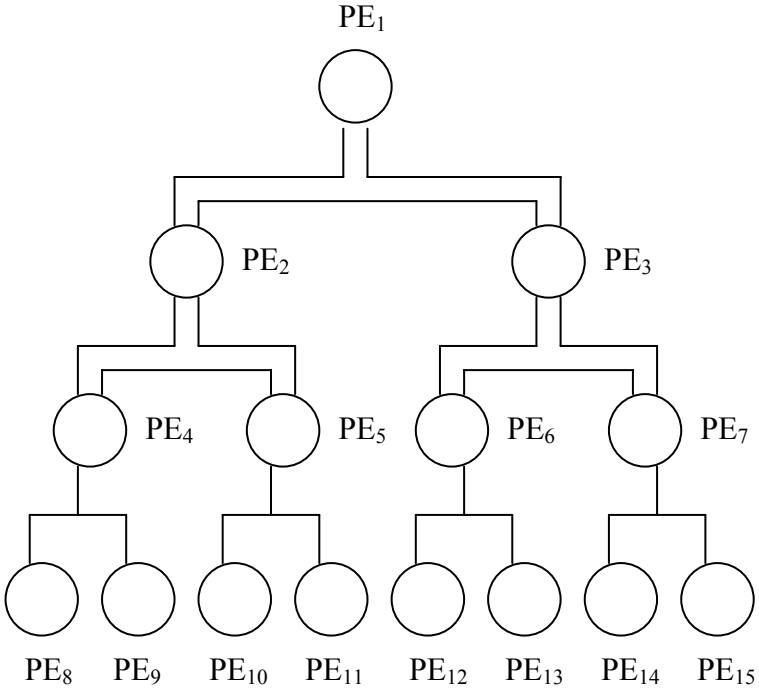


Figure 9: Fat tree

9) Systolic Array: This interconnection network is a type of pipelined array architecture and it is designed for multidimensional flow of data. It is used for implementing fixed algorithms. Systolic array designed for performing matrix multiplication is shown below. All interior nodes have degree 6.

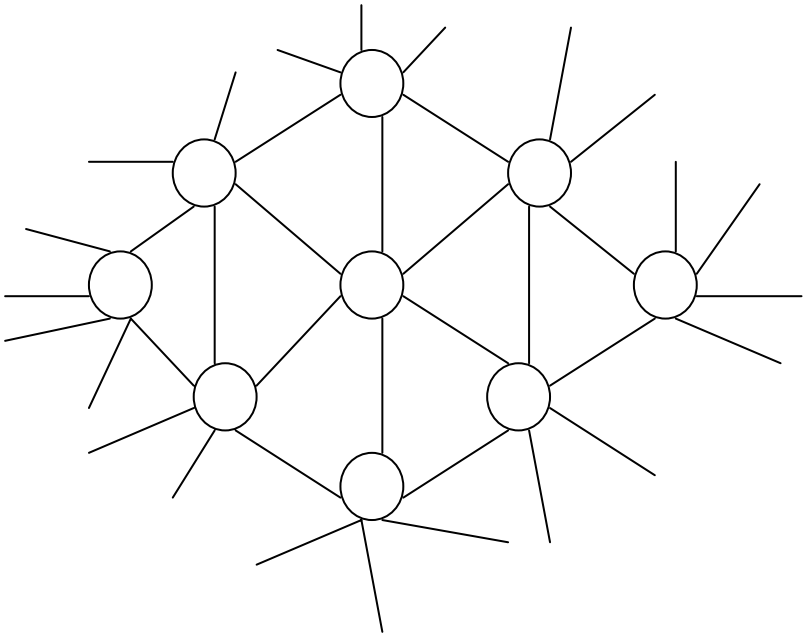


Figure 10: Systolic Array

10) Cube: It is a 3 dimensional interconnection network. In this the PE's are arranged in a cube structure.

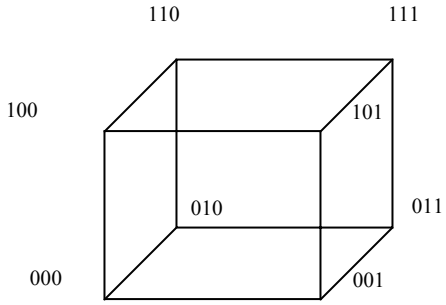


Figure 11: Cube interconnection network

11) Hyper Cube: A Hypercube interconnection network is an extension of cube network. Hypercube interconnection network for n ≥ 3, can be defined recursively as follows:

For n = 3, it cube network in which nodes are assigned number 0, 1, ……,7 in binary. In other words, one of the nodes is assigned a label 000, another one as 001…. and the last node as 111.
Then any node can communicate with any other node if their labels differ in exactly one place, e.g., the node with label 101 may communicate directly with 001, 000 and 111.

For n > 3, a hypercube can be defined recursively as follows:
Take two hypercubes of dimension (n – 1) each having (n –1) bits labels as 00….0, ……11…..1

Next join the two nodes having same labels each (n – 1) -dimension hypercubes and join these nodes. Next prefix '1' the labels of one of the (n – 1) dimensional hypercube and '0' to the labels of the other hypercube. This completes the structure of n-dimensional hypercube. Direct connection is only between that pair of nodes which has a (solid) line connecting the two nodes in the pair.

For n = 4 we draw 4-dimensional hypercube as show in *Figure 12*:

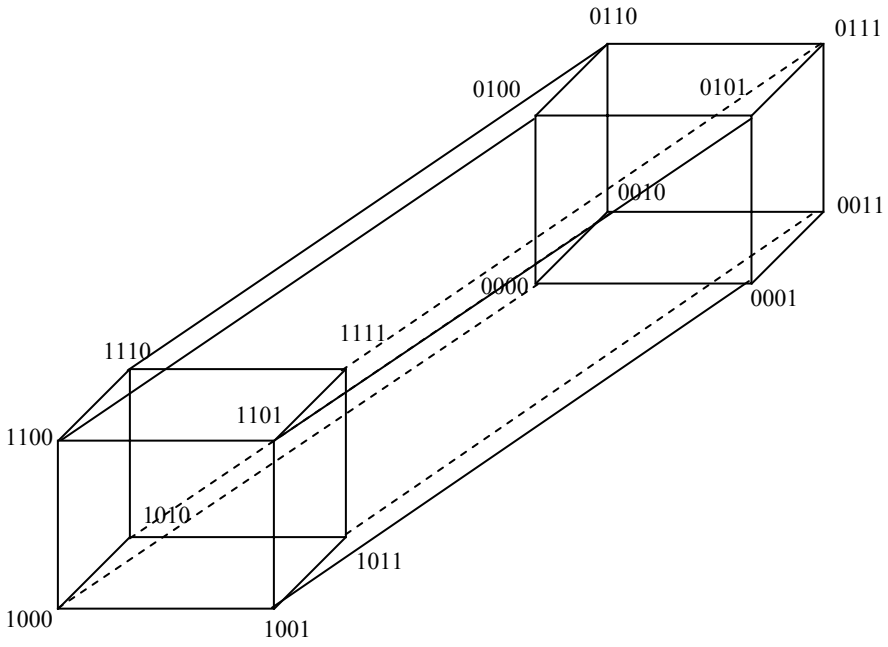


Figure 12: 4-Dimensional hypercube

3.5 CONCEPT OF PERMUTATION NETWORK

In permutation interconnection networks the information exchange requires data transfer from input set of nodes to output set of nodes and possible connections between edges are established by applying various permutations in available links. There are various networks where multiple paths from source to destination are possible. For finding out what the possible routes in such networks are the study of the permutation concept is a must.

Let us look at the basic concepts of permutation with respect to interconnection network. Let us say the network has set of n input nodes and n output nodes.
Permutation P for a network of 5 nodes (i.e., n = 5) is written as follows:

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 1 & 3 & 2 \end{bmatrix}$$

It means node connections are 1↔5, 2↔4, 3↔1, 4↔3, 5↔2.

The connections are shown in the *Figure 13*.

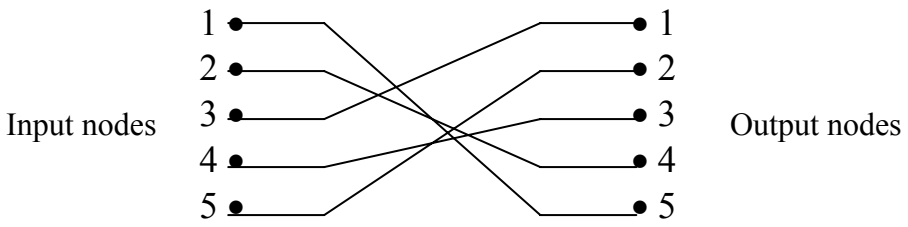


Figure 13: Node-Connections

The other permutation of the same set of nodes may be

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 1 & 4 \end{bmatrix}$$

Which means connections are: 1↔2, 2↔3, 3↔5, 4↔1, and 5↔4
Similarly, other permutations are also possible. The Set of all permutations of a 3-node network will be

$$P= \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

Connection, $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$ indicates connection from node 1 to node1, node 2 to node 2, and node 3 to node 3, hence it has no meaning, so it is dropped.

In these examples, only one set of links exist between input and output nodes and means it is a single stage network. It may be possible that there exist multiple links between input and output (i.e. multistage network). Permutation of all these in a multistage network are called permutation group and these are represented by a cycle e.g. permutation

P= (1,2,3) (4,5) means the network has two groups of input and output nodes, one group consists of nodes 1,2,3 and another group consists of nodes 4,5 and connections are 1→2, 2→3, 3→1, and 4→5. Here group (1,2,3) has period 3 and (4,5) has period 2, collectively these groups has periodicity 3×2=6.

Interconnection from all the possible input nodes to all the output nodes forms the permutation group.

The permutations can be combined. This is called composition operation. In composition operation two or more permutations are applied in sequence, e.g. if P_1 and P_2 are two permutations defined as follows:

$$P_1 = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \quad P_2 \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}$$

The composition of P_1 and P_2 will be

$$P_1 . P_2 = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}$$

$$P_1 . P_2 = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

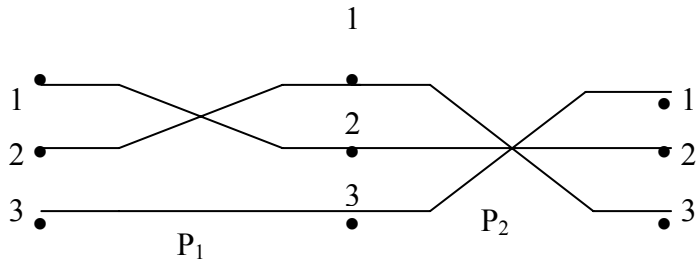


Figure 14 (a)

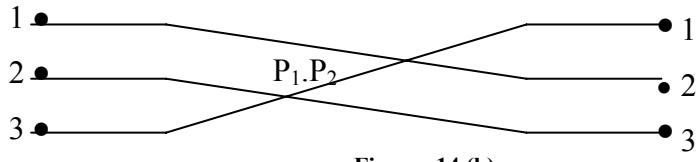


Figure: 14 (b)

Similarly, if $P_3 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 1 & 3 & 5 \end{bmatrix}$ and $P_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 2 & 4 \end{bmatrix}$

then $P_3 . P_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 5 & 4 \end{bmatrix}$

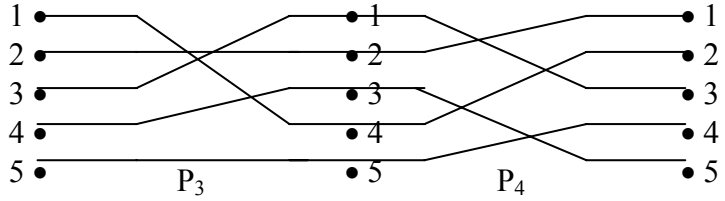


Figure: 15

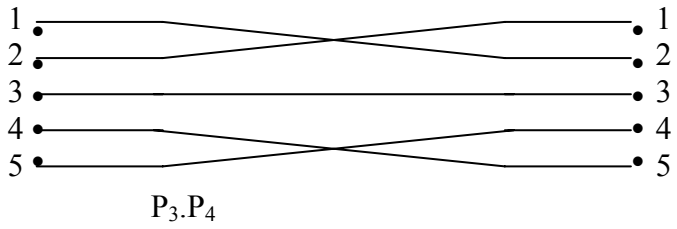


Figure: 16

Composition of those permutations P_1 P_2 are represented in Figures 14 (a) and 14 (b)

There are few permutations of special significance in interconnection network. These permutations are provided by hardware. Now, let us discuss these permutations in detail.

1) **Perfect Shuffle Permutation:** This was suggested by Harold Stone (1971). Consider N objects each represented by n bit number say $X_{n-1,} X_{n-2,} X_0$ (N is chosen such that N = 2n.) The perfect shuffle of these N objects is expressed as

$$X_{n-1,} X_{n-2,} X_0 = X_{n-2,} X_0 \quad X_{n-1.}$$

That, means perfect shuffle is obtained by rotating the address by 1 bit left. e.g. shuffle of 8 objects is shown as

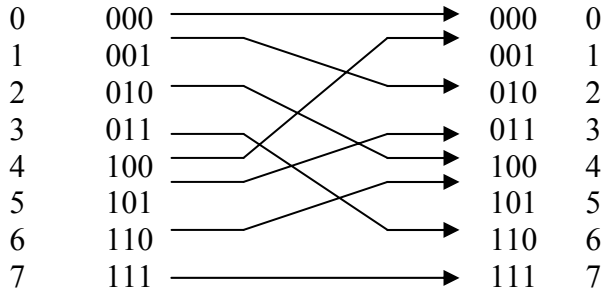


Figure 17: Shuffle of 8 objects

2) **Butterfly permutation:** This permutation is obtained by interchanging the most significant bit in address with least significant bit.

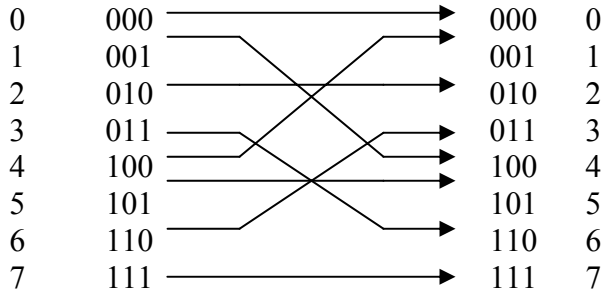


Figure 18: Butterfly permutation

e.g. $X_{n-1,}$ $X_{n-2,}$ and $X_1.X_0 = X_0$ X_{n-2} ……. X_1 X_{n-1}

001↔100, 010↔010
011↔110,

An interconnection network based on this permutation is the butterfly network. A butterfly network is a blocking network and it does not allow an arbitrary connection of

N inputs to N outputs without conflict. The butterfly network is modified in Benz network. The Benz network is a non-blocking network and it is generated by joining two butterfly networks back to back, in such a manner that data flows forward through one and in reverse through the other.

3) **Clos network:** This network was developed by Clos (1953). It is a non-blocking network and provides full connectivity like crossbar network but it requires significantly less number of switches. The organization of Clos network is shown in *Figure 19*:

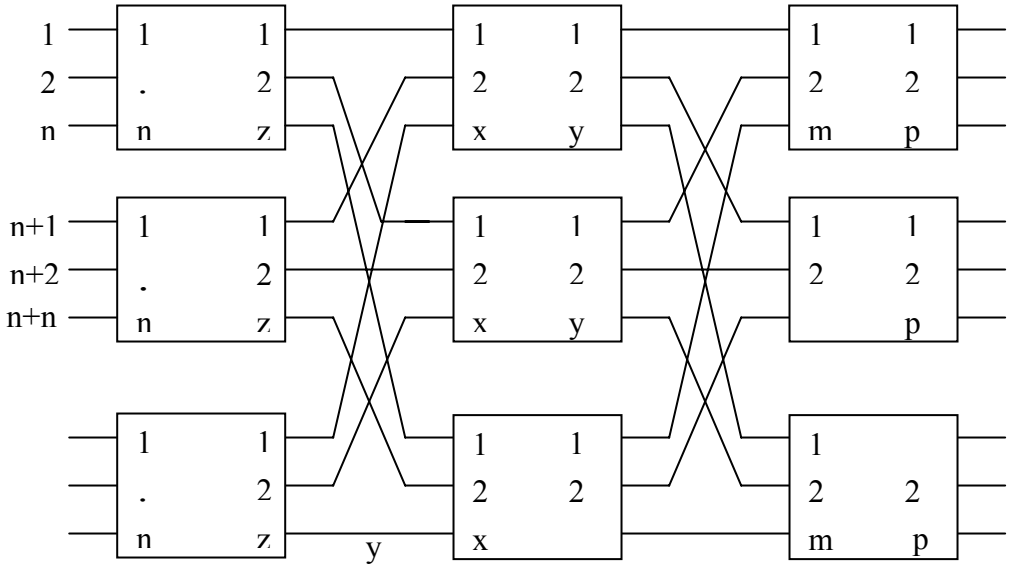


Figure 19: Organisation of Clos network

Consider an I input and O output network
Number N is chosen such that (I= n.x) and (O=p.y).

In Clos network input stage will consist of X switches each having n input lines and z output lines. The last stage will consist of Y switches each having m input lines and p output lines and the middle stage will consist of z crossbar switches, each of size X × Y. To utilize all inputs the value of Z is kept greater than or equal to n and p.

The connection between various stages is made as follows: all outputs of 1st crossbar switch of first stage are joined with 1st input of all switches of middle stage. (i.e., 1st output of first page with 1st middle stage, 2nd output of first stage with 1st input of second switch of middle stage and so on…)

The outputs of second switch of first stage. Stage are joined with 2nd input of various switches of second stage (i.e., 1st output of second switch of 1st stage is joined with 2 input of 1st switch of middle stage and 2nd output of 2nd switch of 1st stage is joined with 2nd input of 2nd switch of middle stage and so on…

Similar connections are made between middle stage and output stage (i.e. outputs of 1st switch of middle stage are connected with 1st input of various switches of third stage.

Permutation matrix of P in the above example the matrix entries will be n

Bens Network: It is a non-blocking network. It is a special type of Clos network where first and last stage consists of 2×2 switches (for n input and m output network it will have n/2 switches of 2 ×2 order and the last stage will have m/2 switch of 2×2 order the middle stage will have two n/2 X m/2 switches. Numbers n and m are assumed to be the power of 2.

Thus, for 16×16 3-stage Bens network first stage and third stage will consist of 8 (2×2) switches and middle stage will consist of 2 switches of size (8×8). The connection of crossbar will be as follows:

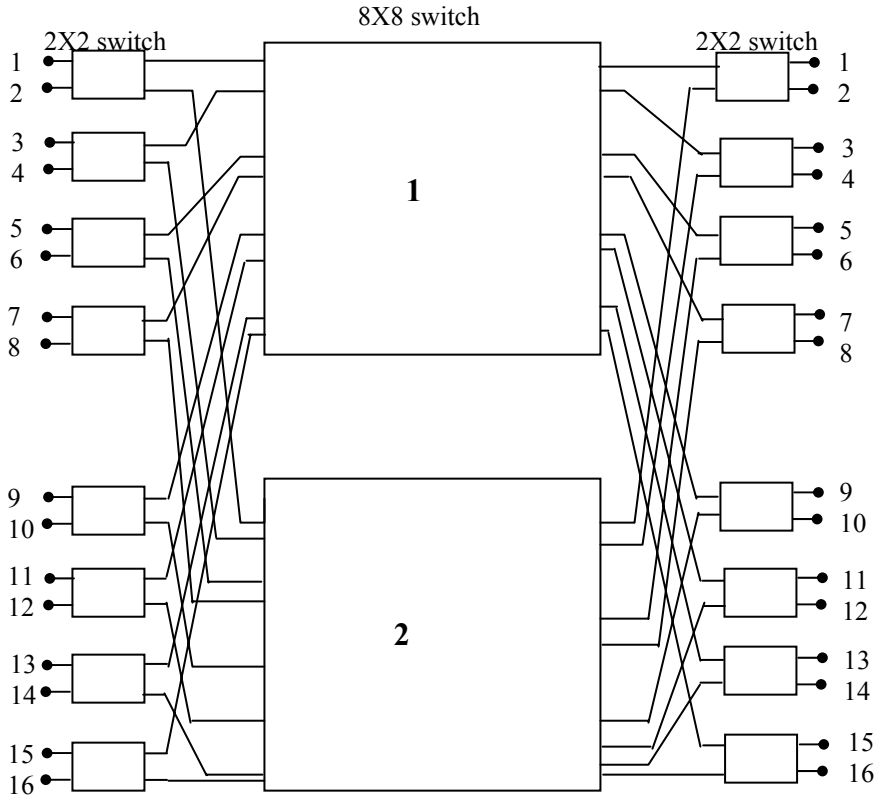


Figure 20: 16X16 3-stage benz network

The switches of various stages provide complete connectivity. Thus by properly configuring the switch any input can be passed to any output.

Consider a Clos network with 3 stages and 3×3 switches in each stage.

Permutation $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 7 & 6 & 2 & 5 & 3 & 9 & 4 & 1 \end{bmatrix}$

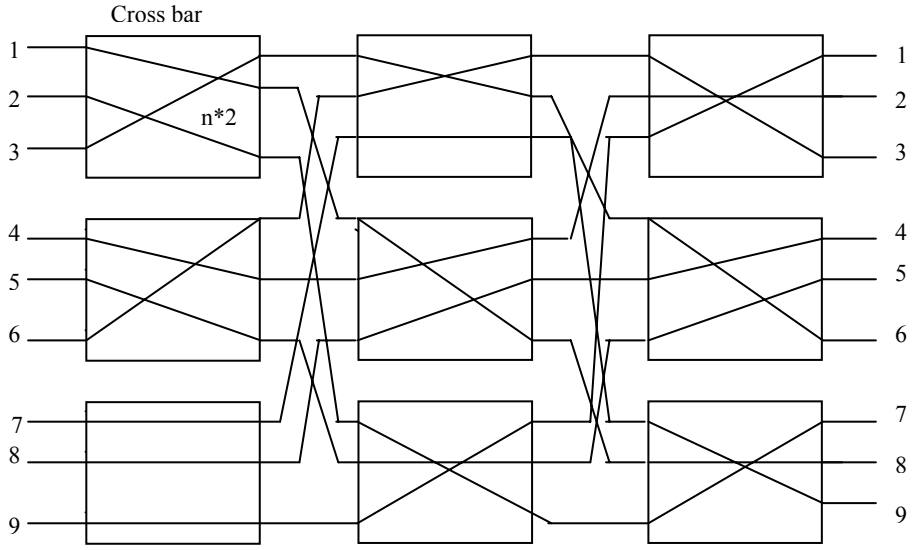


Figure 21: Clos Network

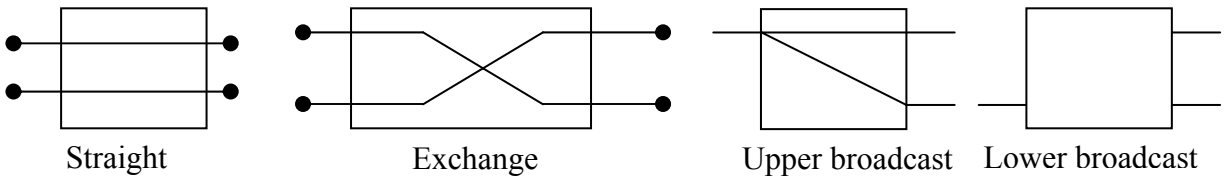
The implementation of this above permutation is shown in *Figure 20*.
This permutation can be represented by the following matrix also,

Input \ Output	1	2	3	4	5	6	7	8	9
1								X	
2							X		
3						X			
4		X							
5					X				
6			X						X
7									
8				X					
9	X								

Figure 22: Permutation representation through Matrix

The upper input of all first stage switches will be connected with respective inputs of 1st middle stage switch, and lower input of all first stage switches will be connected with respective inputs of 2nd switch. Similarly, all outputs of 1st switch of middle stage will be connected as upper input of switches at third stage.
In Benz network to reduce the complexity the middle stage switches can recursively be broken into N/4 × N/4 (then N/8 × M/8), till switch size becomes 2×2.

The connection in a 2×2 switch will either be straight, exchange, lower broadcast or upper broadcast as shown in the Figure.



Switch 2x2

The 8×8 Benz network with all switches replaced by a 2×2 is shown in *Figure 23(a)*:

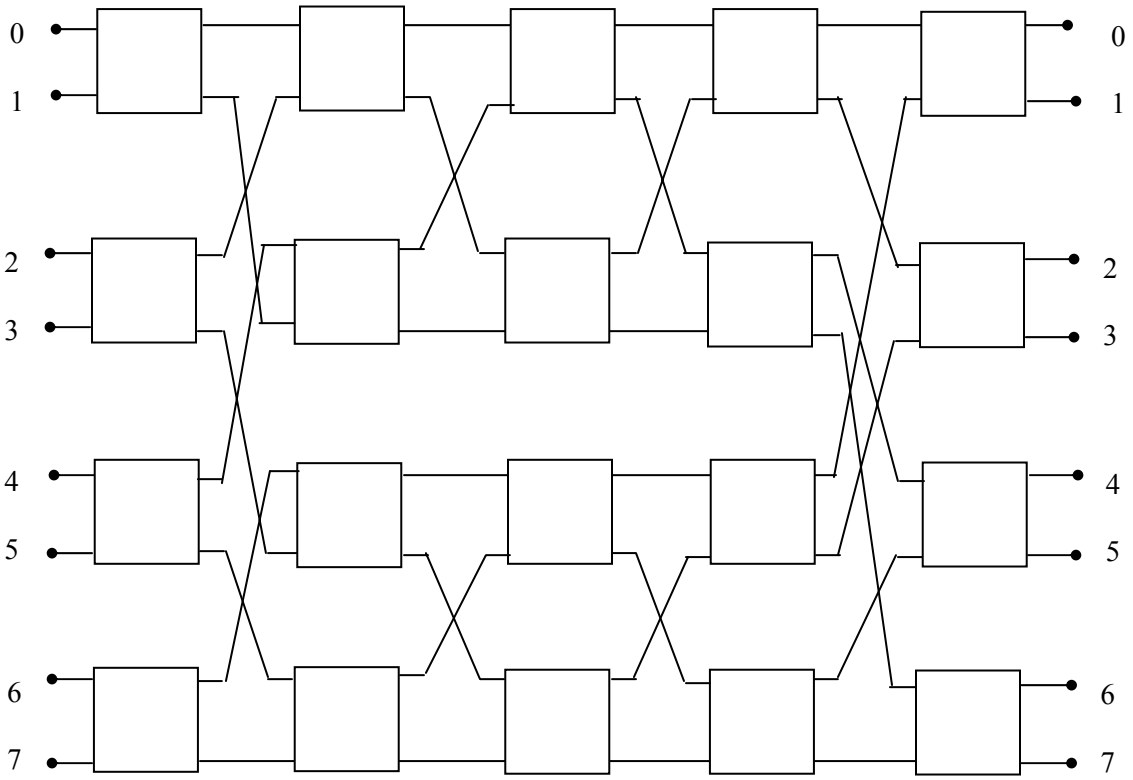


Figure 23 (a): Benz Network

The Bens network connection for permutation

$$P \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 7 & 3 & 4 & 5 & 6 & 2 & 0 \end{bmatrix}$$

will be as follows:-

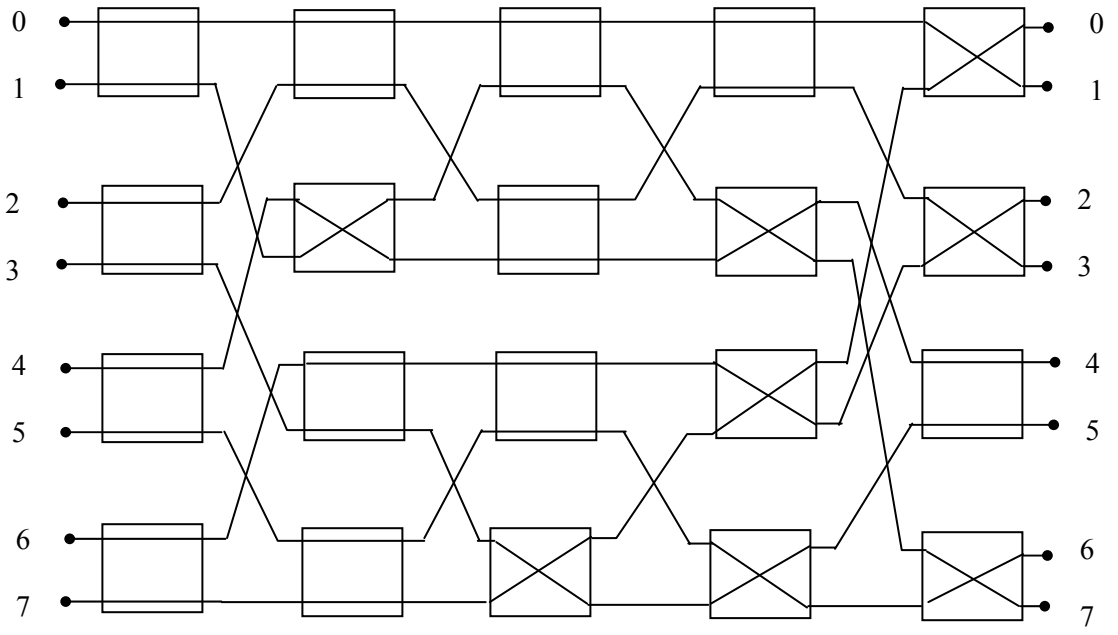


Figure 23 (b): 8X8 BENZ NETWORK OF 4 STAGE

The permutation for P = $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 7 & 6 & 2 & 5 & 3 & 9 & 1 \end{bmatrix}$ will be as follows

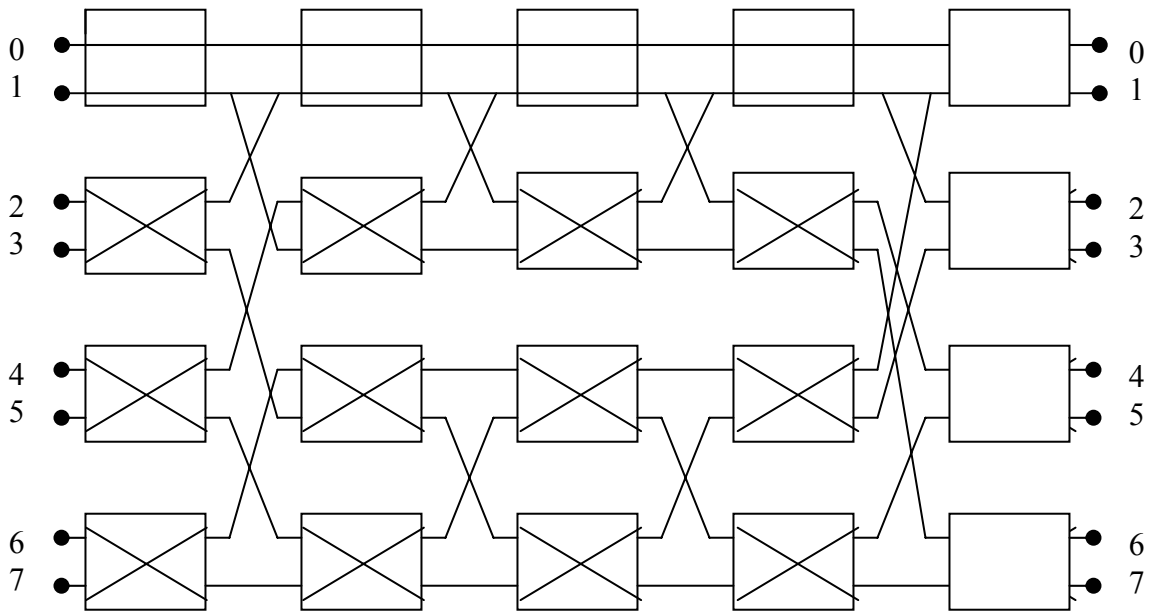


Figure 24: Line number for n

Hardware complexity of Benz Network: - Benz network uses lesser switches and it provides good connectivity. To find hardware complexity of Benz network let us assume that

$$N = 2^n \Rightarrow n = \log_2 N$$

Number of stages in N input Benz network $= 2n-1 = 2 \log_2 N - 1$

Number of 2X2 switches in each stage = N/2.

Total number of cells in network = $(N/2)(2 \log_2 N-1) = N \log_2 N - N/2$

Number of switches in different networks for various inputs is shown in the following table:

Input	No. of cross bars	Switches Clos Net	Benz Net
2	4	9	4
8	64	69	80
64	4096	1536	1408
256	65536	12888	7680

Thus we can analyze from this table that for larger inputs the Benz Network is the best as it has the least number of switches.

Shuffle Exchange Network:- These networks are based on the shuffle and exchange operations discussed earlier.

3.6 PERFORMANCE METRICS

The performance of interconnection networks is measured on the following parameters.

1) **Bandwidth:** It is a measure of maximum transfer rate between two nodes. It is measured in Megabytes per second or Gigabytes per second.

2) **Functionality:** It indicates how interconnection networks supports data routing, interrupt handling, synchronization, request/message combining and coherence.

3) **Latency:** In interconnection networks various nodes may be at different distances depending upon the topology. The network latency refers to the worst-case time delay for a unit message when transferred through the network between farthest nodes.

4) **Scalability:** It refers to the ability of interconnection networks for modular expansion with a scalable performance with increasing machine resources.

5) **Hardware Complexity:** It refers to the cost of hardware logic like wires, connectors, switches, arbiter etc. that are required for implementation of interconnection network.

3.7 SUMMARY

This unit deals with various concepts about interconnection network. The design issues of interconnection network, types of interconnection network, permutation network and performance metrics of the interconnection networks are discussed.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) i) **Node degrees:** Number of edges connected with a node is called node degree. If the edge carries data from the node, it is called out degree and if this carries data into the node it is called in degree.

   ii) **Dynamic connection network:** In dynamic network the interconnection pattern between inputs and outputs can be changed. The interconnection pattern can be reconfigured according to the program demands. Here, instead of fixed connections, the switches or arbiters are used. Examples of such networks are buses, crossbar switches, and multistage networks. The dynamic networks are normally used in shared memory(SM) multiprocessors.

   iii) **Network diameter:** It is the minimum distance between the farthest nodes in a network. The distance is measured in terms of number of distinct hops between any two nodes.

2) Bisection bandwidth of a network is an indicator of robustness of a network in the sense that if the bisection bandwidth is large then there may be more alternative routes between a pair of nodes, any one of the other alternative routes may be chosen. However, the degree of difficulty of dividing a network into smaller networks, is inversely proportional to the bisection bandwidth.

UNIT 4 PARALLEL COMPUTER ARCHITECTURE

Structure **Page Nos.**

4.0 INTRODUCTION

We have discussed the classification of parallel computers and their interconnection networks respectively in units 2 and 3 of this block. In this unit, various parallel architectures are discussed, which are based on the classification of parallel computers considered earlier. The two major parametric considerations in designing a parallel computer architecture are: (i) executing multiple number of instructions in parallel, and (ii) increasing the efficiency of processors. There are various methods by which instructions can be executed in parallel and parallel architectures are based on these methods of executing instructions in parallel. Pipelining is one of the classical and effective methods to increase parallelism where different stages perform repeated functions on different operands. Vector processing is the arithmetic or logical computation applied on vectors whereas in scalar processing only one data item or a pair of data items is processed. Parallel architectures have also been developed based on associative memory organizations. Another idea of improving the processor's speed by having multiple instructions per cycle is known as Superscalar processing. Multithreading for increasing processor utilization has also been used in parallel computer architecture. All the architectures based on these parallel-processing types have been discussed in detail in this unit.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the meaning of Pipeline processing and describe pipeline processing architectures;
- identify the differences between scalar, superscalar and vector processing and their architectures;

- describe architectures based on associative memory organisations, and
- explain the concept of multithreading and its use in parallel computer architecture.

4.2 PIPELINE PROCESSING

Pipelining is a method to realize, overlapped parallelism in the proposed solution of a problem, on a digital computer in an economical way. To understand the concept of pipelining, we need to understand first the concept of assembly lines in an automated production plant where items are assembled from separate parts (stages) and output of one stage becomes the input to another stage. Taking the analogy of assembly lines, pipelining is the method to introduce temporal parallelism in computer operations. Assembly line is the pipeline and the separate parts of the assembly line are different stages through which operands of an operation are passed.

To introduce pipelining in a processor P, the following steps must be followed:

- Sub-divide the input process into a sequence of subtasks. These subtasks will make stages of pipeline, which are also known as segments.
- Each stage S_i of the pipeline according to the subtask will perform some operation on a distinct set of operands.
- When stage S_i has completed its operation, results are passed to the next stage S_{i+1} for the next operation.
- The stage S_i receives a new set of input from previous stage S_{i-1} .

In this way, parallelism in a pipelined processor can be achieved such that m independent operations can be performed simultaneously in m segments as shown in *Figure 1*.

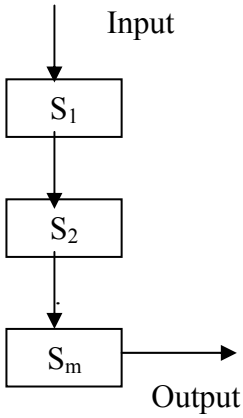


Figure 1: m-Segment Pipeline Processor

The stages or segments are implemented as pure combinational circuits performing arithmetic or logic operations over the data streams flowing through the pipe. Latches are used to separate the stages, which are fast registers to hold intermediate results between the stages as shown in *Figure 2*. Each stage S_i consists of a latch L_i and a processing circuit C_i . The final output is stored in output register R. The flow of data from one stage to another is controlled by a common clock. Thus, in each clock period, one stage transfers its results to another stage.

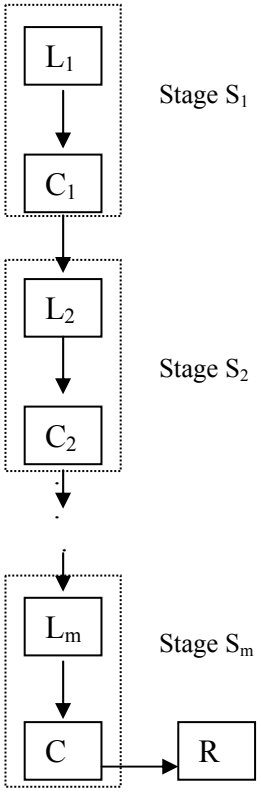


Figure 2: Pipelined Processor

Pipelined Processor: Having discussed pipelining, now we can define a pipeline processor. A pipeline processor can be defined as a processor that consists of a sequence of processing circuits called segments and a stream of operands (data) is passed through the pipeline. In each segment partial processing of the data stream is performed and the final output is received when the stream has passed through the whole pipeline. An operation that can be decomposed into a sequence of well-defined sub tasks is realized through the pipelining concept.

4.2.1 Classification of Pipeline Processors

In this section, we describe various types of pipelining that can be applied in computer operations. These types depend on the following factors:

- Level of Processing
- Pipeline configuration
- Type of Instruction and data

Classification according to level of processing

According to this classification, computer operations are classified as instruction execution and arithmetic operations. Next, we discuss these classes of this classification:

- **Instruction Pipeline:** We know that an instruction cycle may consist of many operations like, fetch opcode, decode opcode, compute operand addresses, fetch operands, and execute instructions. These operations of the instruction execution cycle can be realized through the pipelining concept. Each of these operations forms one stage of a pipeline. The overlapping of execution of the operations through the

pipeline provides a speedup over the normal execution. Thus, the pipelines used for instruction cycle operations are known as *instruction pipelines.*

- **Arithmetic Pipeline:** The complex arithmetic operations like multiplication, and floating point operations consume much of the time of the ALU. These operations can also be pipelined by segmenting the operations of the ALU and as a consequence, high speed performance may be achieved. Thus, the pipelines used for arithmetic operations are known as *arithmetic pipelines*.

Classification according to pipeline configuration:

According to the configuration of a pipeline, the following types are identified under this classification:

- **Unifunction Pipelines**: When a fixed and dedicated function is performed through a pipeline, it is called a Unifunction pipeline.

- **Multifunction Pipelines**: When different functions at different times are performed through the pipeline, this is known as Multifunction pipeline. Multifunction pipelines are reconfigurable at different times according to the operation being performed.

Classification according to type of instruction and data:

According to the types of instruction and data, following types are identified under this classification:

- **Scalar Pipelines**: This type of pipeline processes scalar operands of repeated scalar instructions.

- **Vector Pipelines:** This type of pipeline processes vector instructions over vector operands.

4.2.1.1 Instruction Pipelines

As discussed earlier, the stream of instructions in the instruction execution cycle, can be realized through a pipeline where overlapped execution of different operations are performed. The process of executing the instruction involves the following major steps:

- Fetch the instruction from the main memory
- Decode the instruction
- Fetch the operand
- Execute the decoded instruction

These four steps become the candidates for stages for the pipeline, which we call as instruction pipeline (It is shown in *Figure 3*).

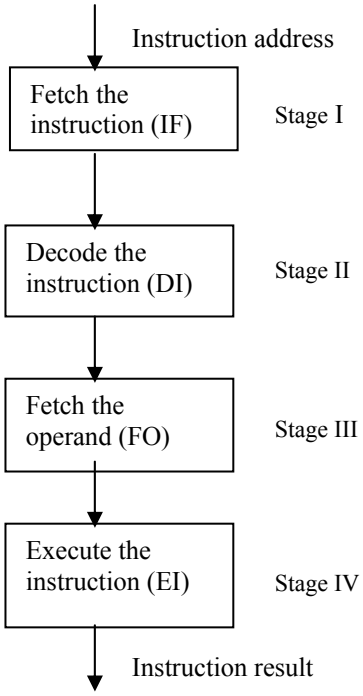


Figure 3: Instruction Pipeline

Since, in the pipelined execution, there is overlapped execution of operations, the four stages of the instruction pipeline will work in the overlapped manner. First, the instruction address is fetched from the memory to the first stage of the pipeline. The first stage fetches the instruction and gives its output to the second stage. While the second stage of the pipeline is decoding the instruction, the first stage gets another input and fetches the next instruction. When the first instruction has been decoded in the second stage, then its output is fed to the third stage. When the third stage is fetching the operand for the first instruction, then the second stage gets the second instruction and the first stage gets input for another instruction and so on. In this way, the pipeline is executing the instruction in an overlapped manner increasing the throughput and speed of execution.

The scenario of these overlapped operations in the instruction pipeline can be illustrated through the space-time diagram. In *Figure 4*, first we show the space-time diagram for non-overlapped execution in a sequential environment and then for the overlapped pipelined environment. It is clear from the two diagrams that in non-overlapped execution, results are achieved only after 4 cycles while in overlapped pipelined execution, after 4 cycles, we are getting output after each cycle. Soon in the instruction pipeline, the instruction cycle has been reduced to ¼ of the sequential execution.

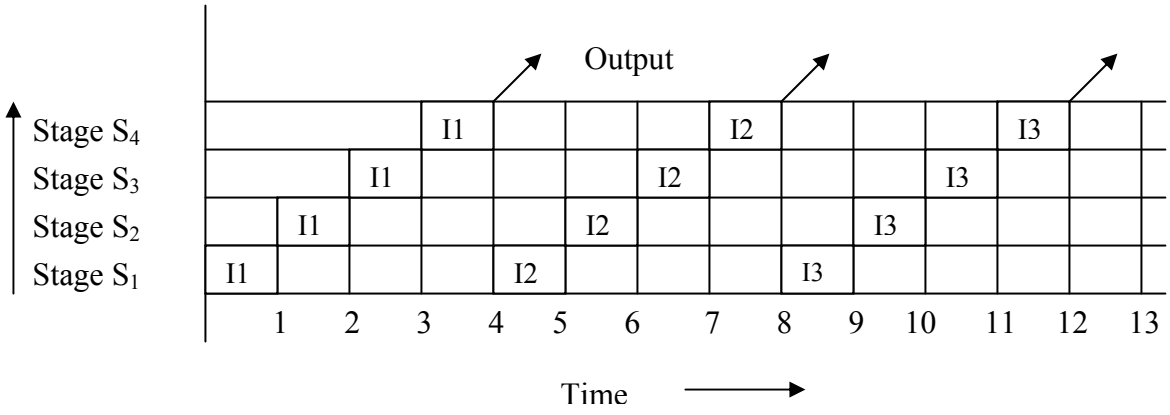


Figure 4(a) Space-time diagram for Non-pipelined Processor

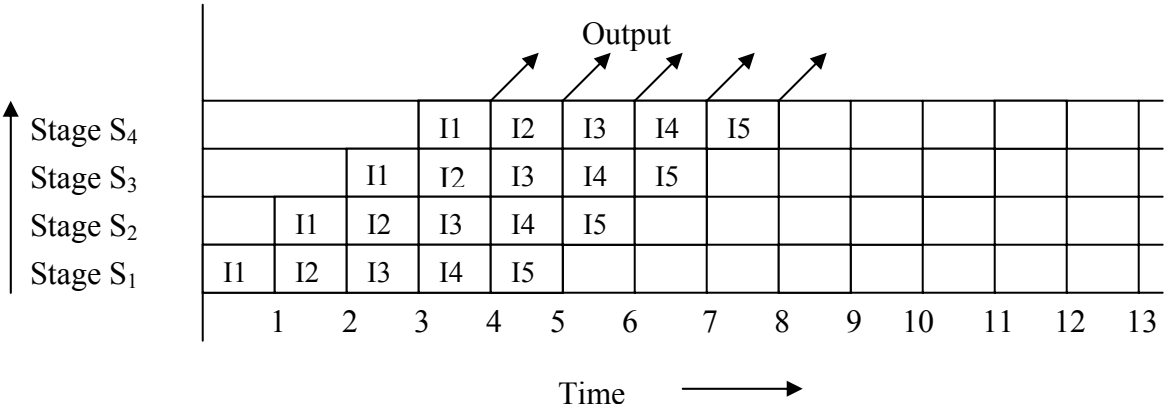


Figure 4(b) Space-time diagram for Overlapped Instruction pipelined Processor

Instruction buffers: For taking the full advantage of pipelining, pipelines should be filled continuously. Therefore, instruction fetch rate should be matched with the pipeline consumption rate. To do this, instruction buffers are used. Instruction buffers in CPU have high speed memory for storing the instructions. The instructions are pre-fetched in the buffer from the main memory. Another alternative for the instruction buffer is the cache memory between the CPU and the main memory. The advantage of cache memory is that it can be used for both instruction and data. But cache requires more complex control logic than the instruction buffer. Some pipelined computers have adopted both.

4.2.1.2 Arithmetic Pipelines

The technique of pipelining can be applied to various complex and slow arithmetic operations to speed up the processing time. The pipelines used for arithmetic computations are called *Arithmetic pipelines*. In this section, we discuss arithmetic pipelines based on arithmetic operations. Arithmetic pipelines are constructed for simple fixed-point and complex floating-point arithmetic operations. These arithmetic operations are well suited to pipelining as these operations can be efficiently partitioned into subtasks for the pipeline stages. For implementing the arithmetic pipelines we generally use following two types of adder:

i) **Carry propagation adder (CPA)**: It adds two numbers such that carries generated in successive digits are propagated.

ii) **Carry save adder (CSA)**: It adds two numbers such that carries generated are not propagated rather these are saved in a carry vector.

Fixed Arithmetic pipelines: We take the example of multiplication of fixed numbers. Two fixed-point numbers are added by the ALU using add and shift operations. This sequential execution makes the multiplication a slow process. If we look at the multiplication process carefully, then we observe that this is the process of adding the multiple copies of shifted multiplicands as show below:

$$X_5 \quad X_4 \quad X_3 \quad X_2 \quad X_1 \quad X_0 = X$$

$$Y_5 \quad Y_4 \quad Y_3 \quad Y_2 \quad Y_1 \quad Y_0 = Y$$

$$X_5Y_0 \; X_4Y_0 \; X_3Y_0 \; X_2Y_0 \; X_1Y_0 \; X_0Y_0 = P_1$$

$$X_5Y_1 \; X_4Y_1 \; X_3Y_1 \; X_2Y_1 \; X_1Y_1 \; X_0Y_1 = P_2$$

$$X_5Y_2 \; X_4Y_2 \; X_3Y_2 \; X_2Y_2 \; X_1Y_2 \; X_0Y_2 = P_3$$

$$X_5Y_3 \; X_4Y_3 \; X_3Y_3 \; X_2Y_3 \; X_1Y_3 \; X_0Y_3 = P_4$$

$$X_5Y_4 \; X_4Y_4 \; X_3Y_4 \; X_2Y_4 \; X_1Y_4 \; X_0Y_4 = P_5$$

$$X_5Y_5 \; X_4Y_5 \; X_3Y_5 \; X_2Y_5 \; X_1Y_5 \; X_0Y_5 = P_6$$

Now, we can identify the following stages for the pipeline:

- The first stage generates the partial product of the numbers, which form the six rows of shifted multiplicands.
- In the second stage, the six numbers are given to the two CSAs merging into four numbers.
- In the third stage, there is a single CSA merging the numbers into 3 numbers.
- In the fourth stage, there is a single number merging three numbers into 2 numbers.
- In the fifth stage, the last two numbers are added through a CPA to get the final product.

These stages have been implemented using CSA tree as shown in *Figure 5*.

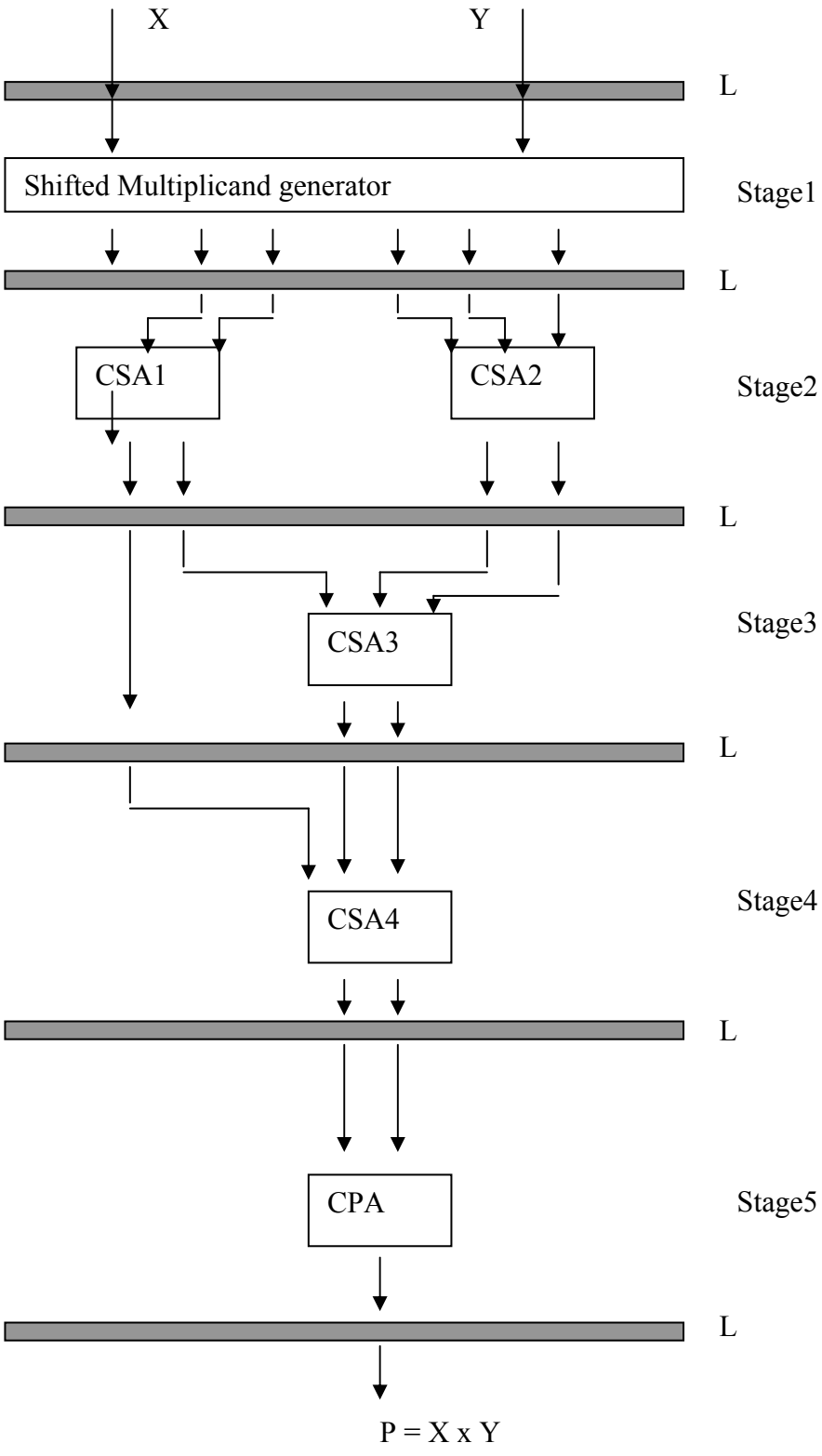


Figure 5: Arithmetic pipeline for Multiplication of two 6-digit fixed numbers

Floating point Arithmetic pipelines: Floating point computations are the best candidates for pipelining. Take the example of addition of two floating point numbers. Following stages are identified for the addition of two floating point numbers:

- First stage will compare the exponents of the two numbers.
- Second stage will look for alignment of mantissas.
- In the third stage, mantissas are added.
- In the last stage, the result is normalized.

These stages are shown in *Figure 6*.

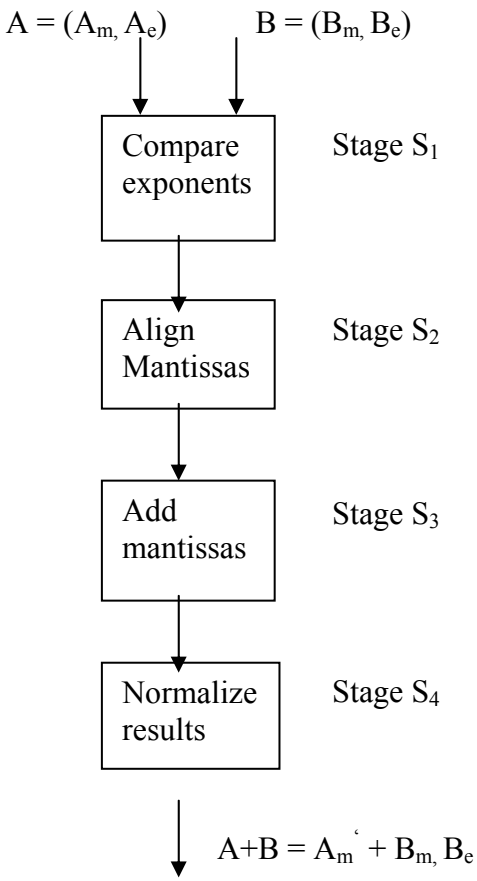


Figure 6: Arithmetic Pipeline for Floating point addition of two numbers

4.2.2 Performance and Issues in Pipelining

Speedup : First we take the speedup factor that is we see how much speed up performance we get through pipelining.

First we take the *ideal case* for measuring the speedup.

Let n be the total number of tasks executed through m stages of pipelines.

Then m stages can process n tasks in clock cycles = m + (n-1)
Time taken to execute without pipelining = m.n

Speedup due to pipelining = m.n/[m +(n-1)].

As n>=∞ , There is speedup of n times over the non-pipelined execution.

Efficiency: The efficiency of a pipeline can be measured as the ratio of busy time span to the total time span including the idle time. Let c be the clock period of the pipeline, the efficiency E can be denoted as:

$$E = (n.\ m.\ c) / m.[m.c + (n-1).c] = n / (m + (n-1)$$

As $n \to \infty$, E becomes 1.

Throughput: Throughput of a pipeline can be defined as the number of results that have been achieved per unit time. It can be denoted as:

$$T = n / [m + (n-1)].\ c = E / c$$

Throughput denotes the computing power of the pipeline.

Maximum speedup, efficiency and throughput are the ideal cases but these are not achieved in the practical cases, as the speedup is limited due to the following factors:

- **Data dependency between successive tasks:** There may be dependencies between the instructions of two tasks used in the pipeline. For example, one instruction cannot be started until the previous instruction returns the results, as both are interdependent. Another instance of data dependency will be when that both instructions try to modify the same data object. These are called *data hazards*.
- **Resource Constraints:** When resources are not available at the time of execution then delays are caused in pipelining. For example, if one common memory is used for both data and instructions and there is need to read/write and fetch the instruction at the same time then only one can be carried out and the other has to wait. Another example is of limited resource like execution unit, which may be busy at the required time.
- **Branch Instructions and Interrupts in the program:** A program is not a straight flow of sequential instructions. There may be branch instructions that alter the normal flow of program, which delays the pipelining execution and affects the performance. Similarly, there are interrupts that postpones the execution of next instruction until the interrupt has been serviced. Branches and the interrupts have damaging effects on the pipelining.

Check Your Progress 1

1) What is the purpose of using latches in a pipelined processor?
   ……
   ……
   ……
   ……
2) Differentiate between instruction pipeline and arithmetic pipeline?
   ……
   ……
   ……
   ……
3) Identify the factors due to which speed of the pipelining is limited?
   ……
   ……
   ……
   ……

4.3 VECTOR PROCESSING

A *vector* is an ordered set of the same type of scalar data items. The scalar item can be a floating pint number, an integer or a logical value. *Vector processing* is the arithmetic or logical computation applied on vectors whereas in scalar processing only one or pair of data is processed. Therefore, vector processing is faster compared to scalar processing. When the scalar code is converted to vector form then it is called v*ectorization*. A *vector processor* is a special coprocessor, which is designed to handle the vector computations.

Vector instructions can be classified as below:

- **Vector-Vector Instructions**: In this type, vector operands are fetched from the vector register and stored in another vector register. These instructions are denoted with the following function mappings:

  $F1 : V \rightarrow V$

  $F2 : V \times V \rightarrow V$

  For example, vector square root is of F1 type and addition of two vectors is of F2.
- **Vector-Scalar Instructions**: In this type, when the combination of scalar and vector are fetched and stored in vector register. These instructions are denoted with the following function mappings:

  $F3 : S \times V \rightarrow V$ where S is the scalar item

  For example, vector-scalar addition or divisions are of F3 type.
- **Vector reduction Instructions**: When operations on vector are being reduced to scalar items as the result, then these are vector reduction instructions. These instructions are denoted with the following function mappings:

  $F4 : V \rightarrow S$

  $F5 : V \times V \rightarrow S$

  For example, finding the maximum, minimum and summation of all the elements of vector are of the type F4. The dot product of two vectors is generated by F5.

- **Vector-Memory Instructions**: When vector operations with memory M are performed then these are vector-memory instructions. These instructions are denoted with the following function mappings:

  $F6 : M \rightarrow V$

  $F7 : V \rightarrow V$

  For example, vector load is of type F6 and vector store operation is of F7.

Vector Processing with Pipelining: Since in vector processing, vector instructions perform the same computation on different data operands repeatedly, vector processing is most suitable for pipelining. Vector processors with pipelines are designed to handle vectors of varying length n where n is the length of vector. A vector processor performs better if length of vector is larger. But large values of n causes the problem in storage of vectors and there is difficulty in moving the vectors to and from the pipelines.

Pipeline Vector processors adopt the following two architectural configurations for this problem as discussed below:

- ***Memory-to-Memory Architecture***: The pipelines can access vector operands, intermediate and final results directly in the main memory. This requires the higher memory bandwidth. Moreover, the information of the base address, the offset and vector length should be specified for transferring the data streams between the main memory and pipelines. STAR-100 and TI-ASC computers have adopted this architecture for vector instructions.

- ***Register*-to-Register Architecture**: In this organization, operands and results are accessed indirectly from the main memory through the scalar or vector registers. The vectors which are required currently can be stored in the CPU registers. Cray-1 computer adopts this architecture for the vector instructions and its CPY contains 8 vector registers, each register capable of storing a 64 element vector where one element is of 8 bytes.

Efficiency of Vector Processing over Scalar Processing:

As we know, a sequential computer processes scalar operands one at a time. Therefore if we have to process a vector of length n through the sequential computer then the vector must be broken into n scalar steps and executed one by one.

For example, consider the following vector addition:

$$\mathbf{A + B \longrightarrow C}$$

The vectors are of length 500. This operation through the sequential computer can be specified by 500 add instructions as given below:

```
C[1] = A[1] + B[1]
C[2] = A[2] + B[2]
----------------------
C[500] = A[500] + B[500]
```

If we perform the same operation through a pipelined-vector computer then it does not break the vectors in 500 add statements. Because a vector processor has the set of vector instructions that allow the operations to be specified in one vector instruction as:

$$\mathbf{A\ (1:500) + B\ (1:500) \longrightarrow C\ (1:500)}$$

Each vector operation may be broken internally in scalar operations but they are executed in parallel which results in mush faster execution as compared to sequential computer.

Thus, the advantage of adopting vector processing over scalar processing is that it eliminates the overhead caused by the loop control required in a sequential computer.

4.4 ARRAY PROCESSING

We have seen that for performing vector operations, the pipelining concept has been used. There is another method for vector operations. If we have an array of n processing elements (PEs) i.e., multiple ALUs for storing multiple operands of the vector, then an n instruction, for example, vector addition, is broadcast to all PEs such that they add all

operands of the vector at the same time. That means all PEs will perform computation in parallel. All PEs are synchronised under one control unit. This organisation of synchronous array of PEs for vector operations is called *Array Processor.* The organisation is same as in SIMD which we studied in unit 2. An array processor can handle one instruction and multiple data streams as we have seen in case of SIMD organisation. Therefore, array processors are also called *SIMD array computers*.

The organisation of an array processor is shown in *Figure 7*. The following components are organised in an array processor:

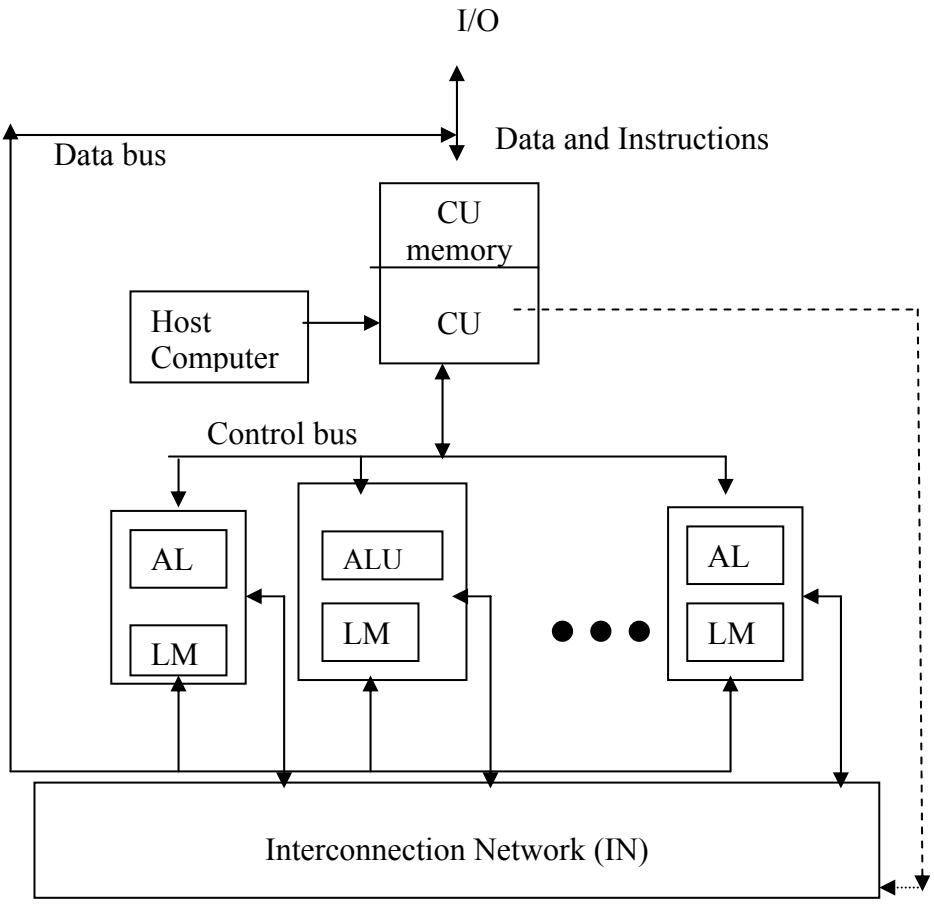


Figure 7: Organisation of SIMD Array Processor

Control Unit (CU) : All PEs are under the control of one control unit. CU controls the inter communication between the PEs. There is a local memory of CU also called CY memory. The user programs are loaded into the CU memory. The vector instructions in the program are decoded by CU and broadcast to the array of PEs. Instruction fetch and decoding is done by the CU only.

Processing elements (PEs) : Each processing element consists of ALU, its registers and a local memory for storage of distributed data. These PEs have been interconnected via an interconnection network. All PEs receive the instructions from the control unit and the different component operands are fetched from their local memory. Thus, all PEs perform the same function synchronously in a lock-step fashion under the control of the CU.

It may be possible that all PEs need not participate in the execution of a vector instruction. Therefore, it is required to adopt a masking scheme to control the status of each PE. A

masking vector is used to control the status of all PEs such that only enabled PEs are allowed to participate in the execution and others are disabled.

Interconnection Network (IN): IN performs data exchange among the PEs, data routing and manipulation functions. This IN is under the control of CU.

Host Computer: An array processor may be attached to a host computer through the control unit. The purpose of the host computer is to broadcast a sequence of vector instructions through CU to the PEs. Thus, the host computer is a general-purpose machine that acts as a manager of the entire system.

Array processors are special purpose computers which have been adopted for the following:

- various scientific applications,
- matrix algebra,
- matrix eigen value calculations,
- real-time scene analysis

SIMD array processor on the large scale has been developed by NASA for earth resources satellite image processing. This computer has been named *Massively parallel processor* (MPP) because it contains 16,384 processors that work in parallel. MPP provides real-time time varying scene analysis.

However, array processors are not commercially popular and are not commonly used. The reasons are that array processors are difficult to program compared to pipelining and there is problem in vectorization.

4.4.1 Associative Array Processing

Consider that a table or a list of record is stored in the memory and you want to find some information in that list. For example, the list consists of three fields as shown below:

Name	ID Number	Age
Sumit	234	23
Ramesh	136	26
Ravi	97	35

Suppose now that we want to find the ID number and age of Ravi. If we use conventional RAM then it is necessary to give the exact physical address of entry related to Ravi in the instruction access the entry such as:

READ ROW 3

Another alternative idea is that we search the whole list using the Name field as an address in the instruction such as:

READ NAME = RAVI

Again with serial access memory this option can be implemented easily but it is a very slow process. An ***associative memory*** helps at this point and simultaneously examines all the entries in the list and returns the desired list very quickly.
SIMD array computers have been developed with *associative memory*. An associative memory is content addressable memory, by which it is meant that multiple memory words

are accessible in parallel. The parallel accessing feature also support parallel search and parallel compare. This capability can be used in many applications such as:

- Storage and retrieval of databases which are changing rapidly
- Radar signal tracking
- Image processing
- Artificial Intelligence

The inherent parallelism feature of this memory has great advantages and impact in parallel computer architecture. The associative memory is costly compared to RAM. The array processor built with associative memory is called *Associative array processor.* In this section, we describe some categories of associative array processor. Types of associative processors are based on the organisation of associative memory. Therefore, first we discuss about the associative memory organisation.

Associative Memory Organisations

The associative memory is organised in *w* words with *b* bits per word. In w x b array, each bit is called a *cell*. Each cell is made up of a flip-flop that contains some comparison logic gates for pattern match and read-write operations. Therefore, it is possible to read or write in parallel due to this logic structure. A group of bit cells of all the words at the same position in a vertical column is called *bit slice* as shown in *Figure 8*.

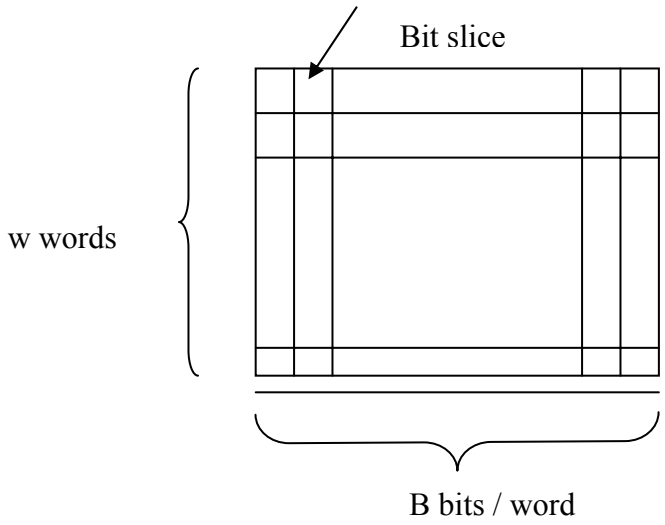


Figure 8: Associative memory

In the organisation of an associative memory, following registers are used:

- *Comparand Register* (C): This register is used to hold the operands, which are being searched for, or being compared with.

- *Masking Register* (M): It may be possible that all bit slices are not involved in parallel operations. Masking register is used to enable or disable the bit slices.

- *Indicator* (I) and *Temporary* (T) *Registers*: Indicator register is used to hold the current match patterns and temporary registers are used to hold the previous match patterns.

There are following two methods for organising the associative memory based on bit slices:

- **Bit parallel organisation:** In this organisation all bit slices which are not masked off, participate in the comparison process, i.e., all words are used in parallel.
- **Bit Serial Organisation:** In this organisation, only one bit slice participate in the operation across all the words. The bit slice is selected through an extra logic and control unit. This organisation is slower in speed but requires lesser hardware as compared to bit parallel which is faster.

Types of Associative Processor

Based on the associative memory organisations, we can classify the associative processors into the following categories:

1) **Fully Parallel Associative Processor**: This processor adopts the bit parallel memory organisation. There are two type of this associative processor:

   - **Word Organized associative processor**: In this processor one comparison logic is used with each bit cell of every word and the logical decision is achieved at the output of every word.

   - **Distributed associative processor**: In this processor comparison logic is provided with each character cell of a fixed number of bits or with a group of character cells. This is less complex and therefore less expensive compared to word organized associative processor.

2) **Bit Serial Associative Processor:** When the associative processor adopts bit serial memory organization then it is called bit serial associative processor. Since only one bit slice is involved in the parallel operations, logic is very much reduced and therefore this processor is much less expensive than the fully parallel associative processor.

PEPE is an example of distributed associative processor which was designed as a special purpose computer for performing real time radar tracking in a missile environment. STARAN is an example of a bit serial associative processor which was designed for digital image processing. There is a high cost performance ratio of associative processors. Due to this reason these have not been commercialised and are limited to military applications.

Check Your Progress 2

1) What is the difference between scalar and vector processing?
   ……
   ……
   ……
   ……
2) Identify the types of the following vector processing instructions?
   a) C(I) = A(I) AND B(I)
   b) C(I) = MAX(A(I), B(I))
   c) B(I) = A(I) / S, where S is a scalar item
   d) B(I) <- SIN (A(I))

……
……
……
……

3) What is the purpose of using the comparand and masking register in the associative memory organisation?

……
……
……
……

4.5 SUPERSCALAR PROCESSORS

In scalar processors, only one instruction is executed per cycle. That means only one instruction is issued per cycle and only one instruction is completed. But the speed of the processor can be improved in scalar pipeline processor if multiple instructions instead of one are issued per cycle. This idea of improving the processor's speed by having multiple instructions per cycle is known as *Superscalar processing*. In superscalar processing multiple instructions are issued per cycle and multiple results are generated per cycle. Thus, the basic idea of superscalar processor is to have more instruction level parallelism.

Instruction Issue degree: The main concept in superscalar processing is how many istructions we can issue per cycle. If we can issue k number of instructions per cycle in a superscalar processor, then that processor is called a k-degree superscalar processor. If we want to exploit the full parallelism from a superscalar processor then k instructions must be executable in parallel.

For example, we consider a 2-degree superscalar processor with 4 pipeline stages for instruction cycle, i.e. instruction fetch (IF), decode instruction (DI), fetch the operands (FO), execute the instruction (EI) as shown in *Figure 3*. In this superscalar processor, 2 instructions are issued per cycle as shown in *Figure 9*. Here, 6 instructions in 4 stage pipeline have been executed in 6 clock cycles. Under ideal conditions, after steady state, two instructions are being executed per cycle.

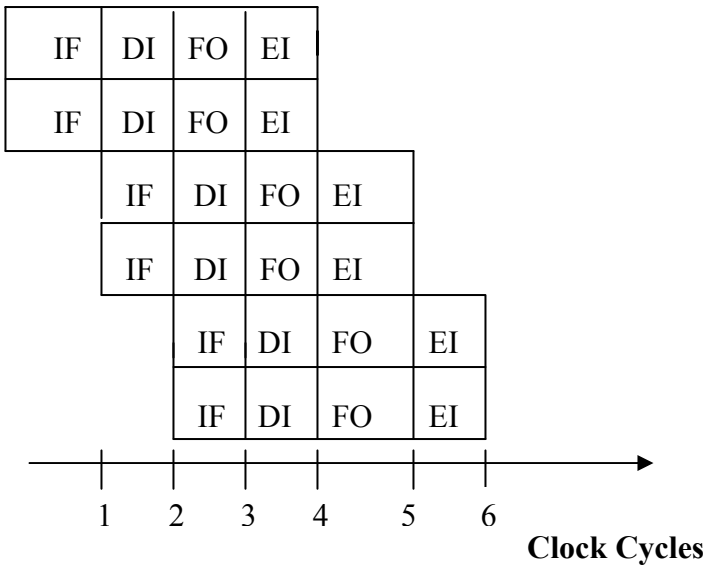


Figure 9: Superscalar Processing of instruction cycle in 4-stage instruction pipeline

For implementing superscalar processing, some special hardware must be provided which is discussed below:

- The requirement of data path is increased with the degree of superscalar processing. Suppose, one instruction size is 32 bit and we are using 2-degree superscalar processor, then 64 data path from the instruction memory is required and 2 instruction registers are also needed.
- Multiple execution units are also required for executing multiple instructions and to avoid resource conflicts.

Data dependency will be increased in superscalar processing if sufficient hardware is not provided. The extra hardware provided is called *hardware machine parallelism*. Hardware parallelism ensures that resource is available in hardware to exploit parallelism. Another alternative is to exploit the *instruction level parallelism* inherent in the code. This is achieved by transforming the source code by an optimizing compiler such that it reduces the dependency and resource conflicts in the resulting code.

Many popular commercial processors have been implemented with superscalar architecture like IBM RS/6000, DEC 21064, MIPS R4000, Power PC, Pentium, etc.

4.6 VLIW ARCHITECTURE

Superscalar architecture was designed to improve the speed of the scalar processor. But it has been realized that it is not easy to implement as we discussed earlier. Following are some problems faced in the superscalar architecture:

- It is required that extra hardware must be provided for hardware parallelism such as instruction registers, decoder and arithmetic units, etc.
- Scheduling of instructions dynamically to reduce the pipeline delays and to keep all processing units busy, is very difficult.

Another alternative to improve the speed of the processor is to exploit a sequence of instructions having no dependency and may require different resources, thus avoiding resource conflicts. The idea is to combine these independent instructions in a compact long word incorporating many operations to be executed simultaneously. That is why; this architecture is called ***very long instruction word (VLIW) architecture***. In fact, long instruction words carry the opcodes of different instructions, which are dispatched to different functional units of the processor. In this way, all the operations to be executed simultaneously by the functional units are synchronized in a VLIW instruction. The size of the VLIW instruction word can be in hundreds of bits. VLIW instructions must be formed by compacting small instruction words of conventional program. The job of compaction in VLIW is done by a compiler. The processor must have the sufficient resources to execute all the operations in VLIW word simultaneously.

For example, one VLIW instruction word is compacted to have load/store operation, floating point addition, floating point multiply, one branch, and one integer arithmetic as shown in *Figure 10*.

Load/Store	FP Add	FP Multiply	Branch	Integer arithmetic

Figure 10: VLIW instruction word

A VLIW processor to support the above instruction word must have the functional components as shown in *Figure 11*. All the functions units have been incorporated according to the VLIW instruction word. All the units in the processor share one common large register file.

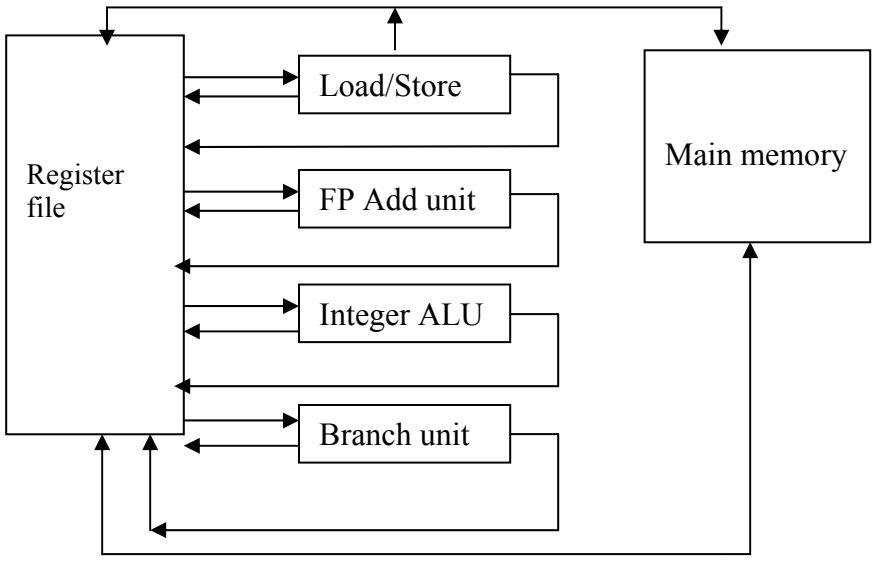


Figure 11: VLIW Processor

Parallelism in instructions and data movement should be completely specified at compile time. But scheduling of branch instructions at compile time is very difficult. To handle branch instructions, *trace scheduling* is adopted. Trace scheduling is based on the prediction of branch decisions with some reliability at compile time. The prediction is based on some heuristics, hints given by the programmer or using profiles of some previous program executions.

4.7 MULTI-THREADED PROCESSORS

In unit 2, we have seen the use of distributed shared memory in parallel computer architecture. But the use of distributed shared memory has the problem of accessing the remote memory, which results in latency problems. This problem increases in case of large-scale multiprocessors like massively parallel processors (MPP).

For example, one processor in a multiprocessor system needs two memory loads of two variables from two remote processors as shown in *Figure 12*. The issuing processor will use these variables simultaneously in one operation. In case of large-scale MPP systems, the following two problems arise:

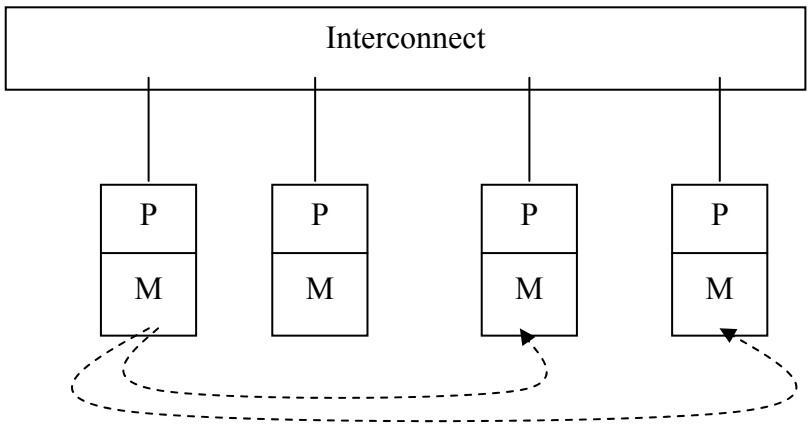


Figure 12: Latency problems in MPP

Remote-load Latency Problem: When one processor needs some remote loading of data from other nodes, then the processor has to wait for these two remote load operations. The longer the time taken in remote loading, the greater will be the latency and idle period of the issuing processor.

Synchronization Latency Problem: If two concurrent processes are performing remote loading, then it is not known by what time two processes will load, as the issuing processor needs two remote memory loads by two processes together for some operation. That means two concurrent processes return the results asynchronously and this causes the synchronization latency for the processor.

Concept of Multithreading: These problems increase in the design of large-scale multiprocessors such as MPP as discussed above. Therefore, a solution for optimizing these latency should be acquired at. The concept of *Multithreading* offers the solution to these problems. When the processor activities are multiplexed among many threads of execution, then problems are not occurring. In single threaded systems, only one thread of execution per process is present. But if we multiplex the activities of process among several threads, then the multithreading concept removes the latency problems.

In the above example, if multithreading is implemented, then one thread can be for issuing a remote load request from one variable and another thread can be for remote load for second variable and third thread can be for another operation for the processor and so on.

Multithreaded Architecture: It is clear now that if we provide many contexts to multiple threads, then processors with multiple contexts are called multithreaded systems. These systems are implemented in a manner similar to multitasking systems. A multithreaded processor will suspend the current context and switch to another. In this way, the processor will be busy most of the time and latency problems will also be optimized. Multithreaded architecture depends on the context switching time between the threads. The switching time should be very less as compared to latency time.

The processor utilization or its efficiency can be measured as:

$U = P / (P + I + S)$

where

P = useful processing time for which processor is busy
I = Idle time when processor is waiting
S = Context switch time used for changing the active thread on the procesor

The objective of any parallel system is to keep U as high as possible. U will be high if I and S are very low or negligible. The idea of multithreading systems is to reduce I such that S is not increasing. If context-switching time is more when compared to idle time, then the purpose of multithreaded systems is lost.

Design issues: To achieve the maximum processor utilization in a multithreaded architecture, the following design issues must be addressed:

- *Context Switching time*: S < I, that means very fast context switching mechanism is needed.
- *Number of Threads*: A large number of threads should be available such that processor switches to an active thread from the idle state.

Check Your Progress 3

1) What is the difference between scalar processing and superscalar processing?
   ……
   ……
   ……
   ……
2) If a superscalar processor of degree 3 is used in 4-stage pipeline instructions, then how many instructions will be executed in 7 clock cycles?
   ……
   ……
   ……
   ……
3) What is the condition for compacting the instruction in a VLIW instruction word?
   ……
   ……
   ……
   ……

4.8 SUMMARY

In this unit, we discuss some of the well-known architecture that exploits inherent parallelism in the data or the problem domain. In section 4.2, we discuss pipeline architecture, which is suitable for executing solutions of problems in the cases when either execution of each of the instructions or operations can be divided into non-overlapping stages. In section 4.3, vector processing, another architecture concurrent execution, is discussed. The vector processing architecture is useful when the same operation at a time, is to be applied to a number of operands of the same type. The vector processing may be achieved through pipelined architecture, if the operation can be divided into a number of non-overlapping stages. Alternatively, vector processing can also be achieved through array processing in which by a large number of processing elements are used. All these PEs perform an identical operation on different components of the vector operand(s). The goal of all these architectures discussed so far is the same — expediting the execution speed by exploiting inherent concurrency in the problem domain in the data. This goal of expediting execution at even higher, speed is attempted to be achieved through three other architectures discussed in next three sections. In section 4.5, we discuss the architecture known as superscalar processing architecture, under which more than one instruction per cycles may be executed. Next, in section 4.6, we discuss VLIW architecture, which is useful when program codes, have a number of chunks of instructions which have no dependency and also, hence or otherwise require different resources. Finally, in section

4.7, another approach viz. Multi-threaded processors approach, of expediting execution is discussed. Through multi-threaded approach, the problem of some type of latencies encountered in some of the architectures discussed earlier may be overcome.

4.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Latches are used to separate the stages, which are fast registers to hold intermediate results between the stages.

2) Instruction pipelines are used to execute the stream of instructions in the instruction execution cycle whereas the pipelines used for arithmetic computations, both fixed and floating point operations, are called arithmetic pipelines.

3) A) Data dependency between successive tasks
   B) Unavailability of resources
   C) Branch instructions and interrupts in the program

Check Your Progress 2

1) Vector processing is the arithmetic or logical computation applied on vectors whereas in scalar processing only a pair of data is processed at a time.

2) a) Vector-vector instruction b) vector-vector instruction c) vector-scalar instruction d) vector-vector instruction.

3) The purpose of comparand register in associative memory organization is to hold the operands which are being searched for or being compared with. Masking register is used to enable or disable the bit slices.

Check Your Progress 3

1) In scalar processing, only one instruction is executed per cycle but when multiple instructions are issued per cycle and multiple results are generated per cycle then it is known as superscalar processing.
2) 12
3) The condition for compacting multiple instructions in a VLIW word is that the processor must have the sufficient resources to execute all the operations in VLIW word simultaneously.

UNIT 1 PARALLEL ALGORITHMS

Structure **Page Nos.**

1.0 INTRODUCTION

An algorithm is defined as a sequence of computational steps required to accomplish a specific task. The algorithm works for a given input and will terminate in a well defined state. The basic conditions of an algorithm are: input, output, definiteness, effectiveness and finiteness. The purpose of the development an algorithm is to solve a general, well specified problem.

A concern while designing an algorithm also pertains to the kind of computer on which the algorithm would be exectued. The two forms of architectures of computers are: sequential computer and parallel computer. Therefore, depending upon the architecture of the computers, we have sequential as well as parallel algorithms.

The algorithms which are executed on the sequential computers simply perform according to sequence of steps for solving a given problem. Such algorithms are known as sequential algorithms.

However, a problem can be solved after dividing it into sub-problems and those in turn are executed in parallel. Later on, the results of the solutions of these subproblems can be combined together and the final solution can be achieved. In such situations, the number of processors required would be more than one and they would be communicating with each other for producing the final output. This environment operates on the parallel computer and the special kind of algorithms called parallel algorithms are designed for these computers. The parallel algorithms depend on the kind of parallel computer they are desinged for. Hence, for a given problem, there would be a need to design the different kinds of parallel algorithms depending upon the kind of parallel architecture.

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. The parallel computers can be represented with the help of various kinds of models such as random access machine (RAM), parallel random access machine (PRAM), Interconnection Networks etc. While designing a parallel algorithm, the computational power of various models can be analysed and compared, parallelism can be involved for a given problem on a specific model after understanding the characteriscitics of a model. The analysis of parallel algorithm on different models assist in determining the best model for a problem after receiving the results in terms of the time and space complexity.

In this unit, we have first discussed the various parameters for analysis of an algorithm. Thereafter, the various kinds of computational models such as combinational circuits etc. have been presented. Subsequently, a few problems have been taken up, e.g., sorting, matrix multiplication etc. and solved using parallel algorithms with the help of various parallel compuational models.

1.1 OBJECTIVES

After studying this unit the learner will be able to understand about the following:

- Analysis of Parallel Algorithms;
- Different Models of Computation;
  - Combinational Circuits
  - Interconnection Networks
  - PRAM
- Sorting Computation, and
- Matrix Computation.

1.2 ANALYSIS OF PARALLEL ALGORITHMS

A generic algorithm is mainly analysed on the basis of the following parameters: the time complexity (execution time) and the space complexity (amount of space required). Usually we give much more importance to time complexity in comparison with space complexity. The subsequent section highlights the criteria of analysing the complexity of parallel algorithms. The fundamental parameters required for the analysis of parallel algorithms are as follow:

- Time Complexity
- The Total Number of Processors Required
- The Cost Involved.

1.2.1 Time Complexity

As it happens, most people who implement algorithms want to know how much of a particular resource (such as time or storage) is required for a given algorithm. The parallel architectures have been designed for improving the computation power of the various algorithms. Thus, the major concern of evaluating an algorithm is the determination of the amount of time required to execute. Usually, the time complexity is calculated on the basis of the total number of steps executed to accomplish the desired output.

The Parallel algorithms usually divide the problem into more symmetrical or asymmetrical subproblems and pass them to many processors and put the results back together at one end. The resource consumption in parallel algorithms is both processor cycles on each processor and also the communication overhead between the processors.

Thus, first in the computation step, the local processor performs an arthmetic and logic operation. Thereafter, the various processors communicate with each other for exchanging messages and/or data. Hence, the time complexity can be calculated on the basis of computational cost and communication cost invloved.

The time complexity of an algorithm varies depending upon the instance of the input for a given problem. For example, the already sorted list (10,17, 19, 21, 22, 33) will consume less amout of time than the reverse order of list (33, 22, 21,19,17,10). The time complexity of an algorithm has been categorised into three forms, viz:

i) Best Case Complexity;
ii) Average Case Complexity; and
iii) Worst Case Complexity.

The best case complexity is the least amount of time required by the algorithm for a given input. The average case complexity is the average running time required by the algorithm for a given input. Similarly, the worst case complexity can be defined as the maximum amount of time required by the algorithm for a given input.

Therefore, the main factors involved for analysing the time complexity depends upon the algorightm, parallel computer model and specific set of inputs. Mostly the size of the input is a function of time complexity of the algorithm. The generic notation for describing the time-complexity of any algorithm is discussed in the subsequent sections.

1.2.1.1 Asymptotic Notations

These notations are used for analysing functions. Suppose we have two functions f(n) and g(n) defined on real numbers,

i) *Theta Θ Notation*: The set Θ(g(n)) consists of all functions f(n), for which there exist positive constants c1,c2 such that f(n) is sandwiched between c1*g(n) and c2*g(n), for sufficiently large values of n. In other words,
$$\Theta(g(n)) = \{ 0 <= c1*g(n) <= f(n) <= c2*g(n) \text{ for all } n >= n_o \}$$

ii) *Big O Notation*: The set O(g(n)) consists of all functions f(n), for which there exists positive constants c such that for sufficiently large values of n, we have 0<= f(n) <= c*g(n). In other words,
$$O(g(n)) = \{ 0 <= f(n) <= c*g(n) \text{ for all } n >= n_o \}$$

iii) *Ω Notation*: The function f(n) belongs to the set Ω (g(n)) if there exists positive constants c such that for sufficiently large values of n,
we have 0<= c*g(n) <=f(n). In other words,
$$O(g(n)) = \{ 0 <= c*g(n) <= f(n) \text{ for all } n >= n_o \}.$$

Suppose we have a function $f(n)= 4n^2 + n$, then the order of function is $O(n^2)$. The asymptotic notations provide information about the lower and upper bounds on complexity of an algorithm with the help of Ω and O notations. For example, in the sorting algorithm the lower bound is Ω (n ln n) and upper bound is O (n ln n). However, problems like matrix multiplication have complexities like $O(n^3)$ to $O(n^{2.38})$. Algorithms

which have similar upper and lower bounds are known as optimal algorithms. Therefore, few sorting algorithms are optimal while matrix multiplication based algorithms are not.

Another method of determining the performance of a parallel algorithm can be carried out after calculating a parameter called "speedup". Speedup can be defined as the ratio of the worst case time complexity of the fastest known sequential algorithm and the worst case running time of the parallel algorithm. Basically, speedup determines the performance improvement of parallel algorithm in comparison to sequential algorithm.

$$Speedup = \frac{Worst\ case\ running\ time\ of\ Sequential\ Algorithm}{Worst\ case\ running\ time\ of\ Parallel\ Algorithm}$$

1.2.2 Number of Processors

One of the other factors that assist in analysis of parallel algorithms is the total number of processors required to deliver a solution to a given problem. Thus, for a given input of size say n, the number of processors required by the parallel algorithm is a function of n, usually denoted by TP (n).

1.2.3 Overall Cost

Finally, the total cost of the algorithm is a product of time complexity of the parallel algorithm and the total number of processors required for computation.

*Cost = Time Complexity * Total Number of Processors*

The other form of defining the cost is that it specifies the total number of steps executed collectively by the n number of processors, i.e., *summation of steps*. Another term related with the analysis of the parallel algorithms is *efficiency* of the algorithm. It is defined as the ratio of the worst case running time of the best sequential algorithm and the cost of the parallel algorithm. The efficiency would be mostly less than or equal to 1. In a situation, if efficiency is greater than 1 then it means that the sequential algorithm is faster than the parallel algorithm.

$$Efficiency = \frac{Worst\ case\ running\ time\ of\ Sequential\ Algorithm}{Cost\ of\ Parallel\ Algorithm}$$

1.3 DIFFERENT MODELS OF COMPUTATION

There are various computational models for representing the parallel computers. In this section, we discuss various models. These models would provide a platform for the designing as well as the analysis of the parallel algorithms.

1.3.1 Combinational Circuits

Combinational Circuit is one of the models for parallel computers. In interconnection networks, various processors communicate with each other directly and do not require a shared memory in between. Basically, combinational circuit (cc) is a connected arrangement of logic gates with a set of m input lines and a set of n output lines as shown in *Figure 1*. The combinational circuits are mainly made up of various interconnected components arranged in the form known as *stages* as shown in *Figure 2*.

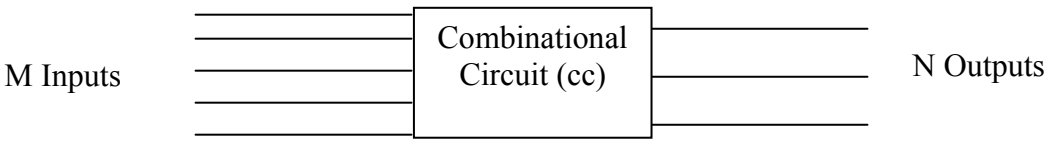


Figure 1: Combinational circuit

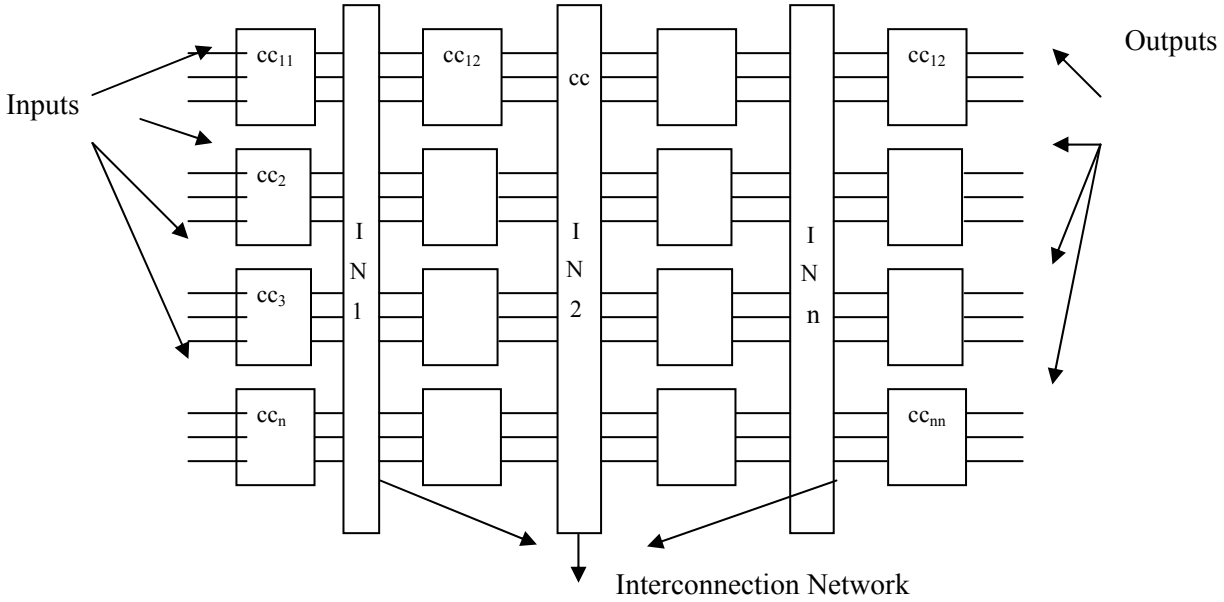


Figure 2: Detailed combinational circuit

It may be noted here that there is no feedback control employed in combinational circuits. There are few terminologies followed in the context of combinational circuits such as fan in and fan out. Fan in signifies the number of input lines attached to each device and fan out signifies the number of output lines. In *Figure 2*, the fan in is 3 and fan out is also 3. The following parameters are used for analysing a combinational circuit:

1) *Depth:* It means that the total number of stages used in the combinational circuit starting from the input lines to the output lines. For example, in the depth is 4, as there are four different stages attached to a interconnection network. The other form of interpretation of depth can be that it represents the worst case time complexity of solving a problem as input is given at the initial input lines and data is transferred between various stages through the interconnection network and at the end reaches the output lines.

2) *Width:* It represents the total number of devices attached for a particular stage. For example in Figure 2, there are 4 components attached to the interconnection network. It means that the width is 4.

3) *Size:* It represents the total count of devices used in the complete combinational circuit. For example, in Figure 2, the size of combinational circuit is 16 i.e. (width * depth).

1.4 PARALLEL RANDOM ACCESS MACHINES (PRAM)

PRAM is one of the models used for designing the parallel algorithm as shown in *Figure 3*. The PRAM model contains the following components:

i) A set of identical type of processors say P_1, P_2, P_3 …P_n.

ii) It contains a single shared memory module being shared by all the N processors. As the processors cannot communicate with each other directly, shared memory acts as a communication medium for the processors.

iii) In order to connect the N processor with the single shared memory, a component called Memory Access Unit (MAU) is used for accessing the shared memory.

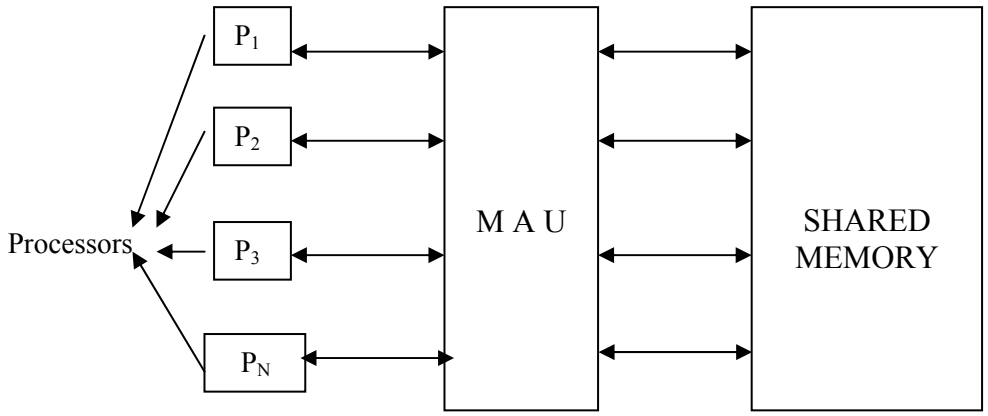


Figure 3: PRAM Model

Following steps are followed by a PRAM model while executing an algorithm:

i) *Read phase:* First, the N processors simultaneously read data from N different memory locations of the shared memory and subsequently store the read data into its local registers.

ii) *Compute phase:* Thereafter, these N processors perform the arithmetic or logical operation on the data stored in their local registers.

iii) *Write phase:* Finally, the N processors parallel write the computed values from their local registers into the N memory locations of the shared memory.

1.5 INTERCONNECTION NETWORKS

As in PRAM, there was no direct communication medium between the processors, thus another model known as interconnection networks have been designed. In the interconnection networks, the N processors can communicate with each other through direct links. In the interconnection networks, each processor has an independent local memory.

☞ **Check Your Progress 1**

1) Which of the following model of computation requires a shared memory?
   1) PRAM
   2) RAM
   3) Interconnection Networks
   4) Combinational Circuits

2) Which of the following model of computation has direct link between processors?
   1) PRAM
   2) RAM
   3) Interconnection Networks
   4) Combinational Circuits

3) What does the term width depth in combinational circuits mean?
   1) Cost
   2) Running Time
   3) Maximum number of components in a given stage
   4) Total Number of stages

4) Explain the concept of analysis of parallel algorithms.

……
……
……
……

1.6 SORTING

The term sorting means arranging elements of a given set of elements, in a specific order i.e., ascending order / descending order / alphabetic order etc. Therefore, sorting is one of the interesting problems encountered in computations for a given data. In the current section, we would be discussing the various kinds of sorting algorithms for different computational models of parallel computers.

The formal representation of sorting problem is as explained: Given a string of m numbers, say X= x_1, x_2, x_3, x_4 ..... x_m and the order of the elements of the string X is initially arbitrary. The solution of the problem is to rearrange the elements of the string X such that the resultant sequence is in an ascending order.

Let us use the combinational circuits for sorting the string.

1.7 COMBINATIONAL CIRCUIT FOR SORTING THE STRING

Each input line of the combinational circuit represents an individual element of the string say x_i and each output line results in the form of a sorted list. In order to achieve the above mentioned task, a comparator is employed for the processing.

Each comparator has two input lines, say a and b, and similarly two output lines, say c and d. Each comparator provides two outputs i.e., c provides maximum of a and b **(max**

(a, b)) and d provides minimum of a and b **(min (a, b))** in comparator InC and DeC it is opposite, as shown in *Figure 4 and 5*.

In general, there are two types of comparators, often known as increasing comparators and decreasing comparators denoted by + BM(n) and – BM(n) where n denotes the number of input lines and output lines of the comparator. The depth of + BM(n) and – BM(n) is log n. These comparators are employed for constructing the circuit of sorting.

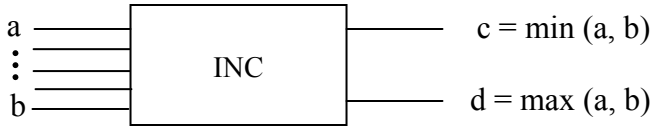


Figure 4 (a) Increasing Comparator, for 2 inputs

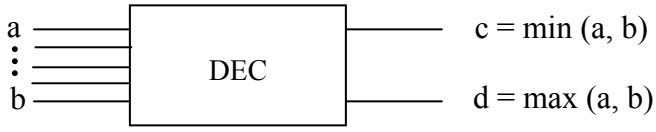


Figure 4 (b) Decreasing Comparator, for 2 inputs

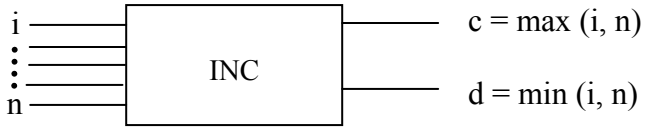


Figure 5 (a): Increasing Comparator, for n inputs

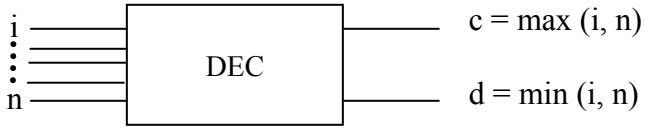


Figure 5 (b): Decreasing Comparator, for n inputs

Now, let us assume a famous sequence known as bitonic sequence and sort out the elements using a combinational circuit consisting of a set of comparators. The property of bitonic sequence is as follows:

Consider a sequence X= $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_{n-1}$ such that condition 1:either $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_i$ is a monotonically increasing sequence and $x_{i+1}, x_{i+2}, \ldots\ldots\ldots\ldots x_{n-1}$ is a monotonically decreasing sequence or condition 2: there exists a cyclic shift of the sequence $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_{n-1}$ such that the resulting sequence satisfies the condition 1.

Let us take a few examples of bitonic sequence:

B1= 4,7,8,9,11,6,3,2,1 is bitonic sequence
B2= 12,15,17,18,19,11,8,7,6,4,5 is bitonic sequence

In order to provide a solution to such a bitonic sequence, various stages of comparators are required. The basic approach followed for solving such a problem is as given:

Let us say we have a bitonic sequence X= x_0, x_1, x_2, x_3, x_4 ..... x_{n-1} with the property that first n/2 elements are monotonically increasing elements are monotonically increasing like $x_0 < x_1 < x_2 < x_3 < x_4 \ldots < x_{n/2-1}$ and other numbers are monotonically decreasing as $x_{n/2} > x_{n/2+1} > \ldots > x_{n-1}$. Thereafter, these patterns are compared with the help of a comparator as follows:

$$Y = \min(x_0, x_{n/2}), \min(x_1, x_{n/2+1}), \min(x_2, x_{n/2+2}), \ldots\ldots\ldots\ldots \text{Min}\ (x_{n/2-1}, x_{n-1})$$
$$Z = \max(x_0, x_{n/2}), \max\ (x_1, x_{n/2+1}), \max\ (x_2, x_{n/2+2}), \ldots\ldots\ldots\ldots \max\ (x_{n/2-1}, x_{n-1})$$

After the above discussed iteration, the two new bitonic sequences are generated and the method is known as bitonic split. Thereafter, a recursive operation on these two bitonic sequences is processed until we are able to achieve the sorted list of elements. The exact algorithm for sorting the bitonic sequence is as follows:

Sort_Bitonic (X)

// Input: N Numbers following the bitonic property
// Output: Sorted List of numbers

1) *The Sequence, i.e. X is transferred on the input lines of the combinational circuit which consists of various set of comparators.*
2) *The sequence X is splitted into two sub-bitonic sequences say, Y and Z, with the help of a method called bitonic split.*
3) *Recursively execute the bitonic split on the sub sequences, i.e. Y and Z, until the size of subsequence reaches to a level of 1.*
4) *The sorted sequence is achieved after this stage on the output lines.*

With the help of a diagram to illustrate the concept of sorting using the comparators.
Example 1: Consider a unsorted list having the element values as
{3,5,8,9,10,12,14,20,95,90,60,40,35,23,18,0}

This list is to be sorted in ascending order. To sort this list, in the first stage comparators of order 2 (i.e. having 2 input and 2 output) will be used. Similarly, 2nd stage will consist of 4, input comparators, 3rd stage 8 input comparator and 4th stage a 16 input comparator.

Let us take an example with the help of a diagram to illustrate the concept of sorting using the comparators (see *Figure 6*).

<table>
<tr><td>3
5</td><td>+BM(2)</td><td>3
5</td><td rowspan="2">+BM(4)</td><td>3
5</td><td rowspan="8"></td><td>3
5</td><td rowspan="16"></td><td>0
3</td></tr>
<tr><td>8
9</td><td>-BM(2)</td><td>9
8</td><td>8
9</td><td>8
9</td><td>5
8</td></tr>
<tr><td>10
12</td><td>+BM(2)</td><td>10
12</td><td rowspan="3">-BM(4)</td><td>20
14</td><td>10
12</td><td>9
10</td></tr>
<tr><td>14
20</td><td>-BM(2)</td><td>20
14</td><td>12
10</td><td>14
20</td><td>12
14</td></tr>
<tr><td>95
90</td><td>+BM(2)</td><td>90
95</td><td rowspan="3">+BM(4)</td><td>40
60</td><td rowspan="7">-BM(8)</td><td>95
90</td><td>18
20</td></tr>
<tr><td>60
40</td><td>-BM(2)</td><td>60
40</td><td>90
95</td><td>60
40</td><td>23
35</td></tr>
<tr><td>35
23</td><td>+BM(2)</td><td>23
35</td><td rowspan="3">-BM(4)</td><td>35
23</td><td>35
23</td><td>40
60</td></tr>
<tr><td>18
0</td><td>-BM(2)</td><td>18
0</td><td>18
0</td><td>18
0</td><td>90
95</td></tr>
</table>

Figure 6: Sorting using Combinational Circuit

Analysis of Sort_Bitonic(X)

The bitonic sorting network requires log n number of stages for performing the task of sorting the numbers. The first n-1 stages of the circuit are able to sort two n/2 numbers and the final stage uses a +BM (n) comparator having the depth of log n. As running time of the sorting is dependent upon the total depth of the circuit, therefore it can be depicted as:

$$D(n) = D(n/2) + \log n$$

Solving the above mentioned recurrence relation

$$D(n) = (\log^2 n + \log n)/2 = O(\log^2 n)$$

Thus, the complexity of solving a sorting algorithm using a combinational circuit is $O(\log^2 n)$.

Another famous sorting algorithm known as merge sort based algorithm can also be depicted / solved with the help of combinational circuit. The basic working of merge sort algorithm is discussed in the next section

1.8 MERGE SORT CIRCUIT

First, divide the given sequence of n numbers into two parts, each consisting of n/2 numbers. Thereafter, recursively split the sequence into two parts until each number acts as an independent sequence. Consequently, the independent numbers are first sorted and recursively merged until a sorted sequence of n numbers is not achieved.

In order to perform the above-mentioned task, there will be two kinds of circuits which would be used in the following manner: the first one for sorting and another one for merging the sorted list of numbers.

Let us discuss the sorting circuit for merge sort algorithm. The sorting Circuit.

Odd-Even Merging Circuit

Let us firstly illustrate the concept of merging two sorted sequences using a odd-even merging circuit. The working of a merging circuit is as follows:

1) Let there be two sorted sequences $A=(a_1, a_2, a_3, a_4 \ldots\ldots a_m)$ and $B=(b_1, b_2, b_3, b_4 \ldots\ldots b_m)$ which are required to be merged.
2) With the help of a merging circuit (m/2,m/2), merge the odd indexed numbers of the two sub sequences i.e. $(a_1, a_3, a_5 \ldots\ldots a_{m-1})$ and $(b_1, b_3, b_5 \ldots\ldots b_{m-1})$ and thus resulting in sorted sequence $(c_1, c_2, c_3 \ldots\ldots c_{m)}$.
3) Thereafter, with the help of a merging circuit (m/2,m/2), merge the even indexed numbers of the two sub sequences i.e. $(a_2, a_4, a_6 \ldots\ldots a_m)$ and $(b_2, b_4, b_6 \ldots\ldots b_m)$ and thus resulting in sorted sequence $(d_1, d_2, d_3 \ldots\ldots d_{m)}$.
4) The final output sequence $O=(o_1, o_2, o_3 \ldots\ldots o_{2m})$ is achieved in the following manner: $o_1 = a_1$ and $o_{2m} = b_m$ .In general the formula is as given below: $o_{2i} = \min(a_{i+1}, b_i)$ and $o_{2I+1} = \max(a_{i+1}, b_i)$ for i=1,2,3,4…….m-1.

Now, let us take an example for merging the two sorted sequences of length 4, i.e., $A=(a_1, a_2, a_3, a_4)$ and $B=(b_1, b_2, b_3, b_4)$. Suppose the numbers of the sequence are A=(4,6,9,10) and B=(2,7,8,12). The circuit of merging the two given sequences is illustrated in *Figure 7*.

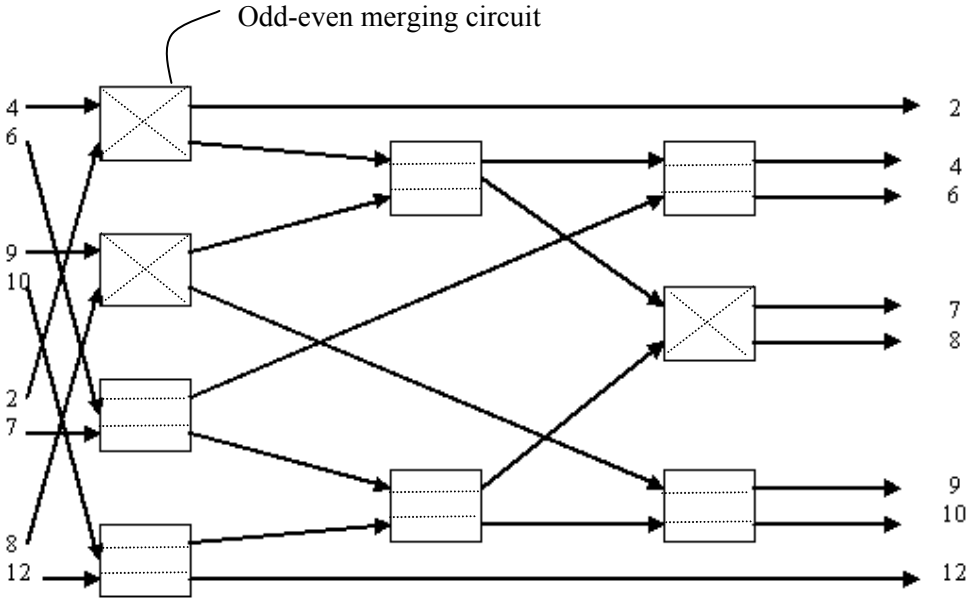


Figure 7: Merging Circuit

Sorting Circuit along with Odd-Even Merging Circuit

As we already know, the merge sort algorithm requires two circuits, i.e. one for merging and another for sorting the sequences. Therefore, the sorting circuit has been derived from the above-discussed merging circuit. The basic steps followed by the circuit are highlighted below:

i) The given input sequence of length n is divided into two sub-sequences of length n/2 each.
ii) The two sub sequences are recursively sorted.
iii) The two sorted sub sequences are merged (n/2,n/2) using a merging circuit in order to finally get the sorted sequence of length n.

Now, let us take an example for sorting the n numbers say 4,2,10,12 8,7,6,9. The circuit of sorting + merging given sequence is illustrated in *Figure 8*.

Analysis of Merge Sort

i) The width of the sorting + merging circuit is equal to the maximum number of devices required in a stage is O(n/2). As in the above figure the maximum number of devices for a given stage is 4 which is 8/2 or n/2.
ii) The circuit contains two sorting circuits for sorting sequences of length n/2 and thereafter one merging circuit for merging of the two sorted sub sequences (see stage 4^{th} in the above figure). Let the functions Ts and Tm denote the time complexity of sorting and merging in terms of its depth.

The Ts can be calculated as follows:

$$Ts(n) = Ts(n/2) + Tm(n/2)$$

$$Ts(n) = Ts(n/2) + \log(n/2) ,$$

Therefore, Ts (n) is equal to $O(\log^2 n)$.

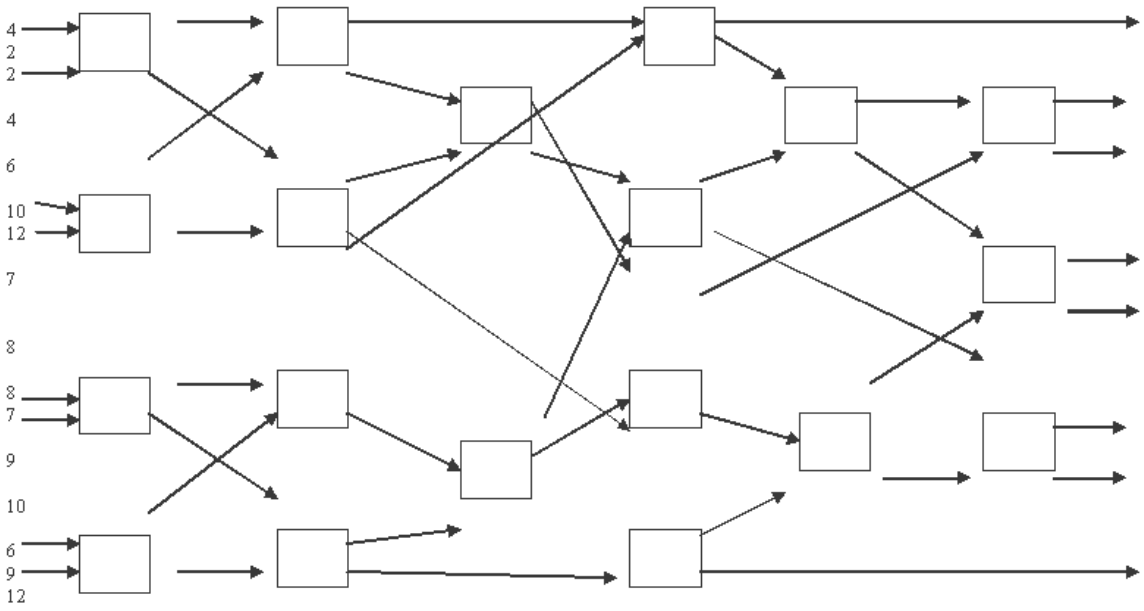


Figure 8: Sorting + Merging Circuit

1.9 SORTING USING INTERCONNECTION NETWORKS

The combinational circuits use the comparators for comparing the numbers and storing them on the basis of minimum and maximum functions. Similarly, in the interconnection networks the two processors perform the computation of minimum and maximum functions in the following way:

Let us consider there are two processors p_i and p_j. Each of these processors has been given as input an element of the sequence, say e_i and e_j. Now, the processor p_i sends the element e_i to p_j and consequently processor p_j sends e_j to p_i. Thereafter, processor p_i calculates the minimum of e_i and e_j i.e., min (e_i,e_j) and processor p_j calculates the maximum of e_i and e_j, i.e. max (e_i,e_j). The above method is known as compare-exchange and it has been depicted in the Figure 9.

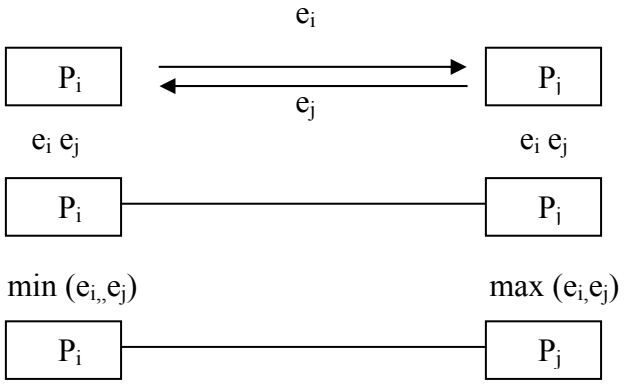


Figure 9: Illustration of Exchange-cum-Comparison in interconnection networks

The sorting problem selected is bubble sort and the interconnection network can be depicted as n processors interconnected with each other in the form of a linear array as

shown in *Figure 10*. The technique adopted for solving the bubble sort is known as odd-even transposition. Assume an input sequence is B=($b_1, b_2, b_3, b_4 \ldots\ldots b_n$) and each number is assigned to a specific processor. In the odd-even transposition, the sorting is performed with the help of two phases called odd phase and even phase. In the odd phase, the elements stored in (p_1, p_2), (p_3, p_4), (p_5, p_6)…… (p_{n-1}, p_n) are compared according to the Figure and subsequently exchanged if required i.e. if they are out of order. In the even phase, the elements stored in (p_2, p_3), (p_4, p_5), (p_6, p_7)…… (p_{n-2}, p_{n-1}) are compared according to the Figure and subsequently exchanged if required, i.e. if they are out of order. Remember, in the even phase the elements stored in p_1 and p_n are not compared and exchanged. The total number of phases required for sorting the numbers is n i.e. n/2 odd phases and n/2 even phases. The algorithmic representation of the above discussed odd-even transposition is given below:

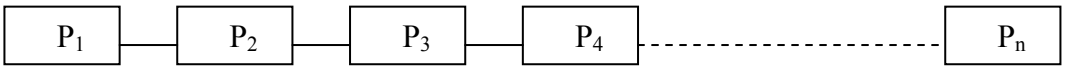


Figure 10: Interconnection network in the form of a Linear Array

Algorithm: Odd-Even Transposition

```
//Input: N numbers that are in the unsorted form
//Assume that element bi is assigned to pi

for I=1 to N
   {
   If (I%2 != 0) //i.e Odd phase
        {
        For j=1,3,5,7,……2*n/2-1
              {
                  Apply compare-exchange(Pj, Pj+1)  //Operation is performed in parallel
              }
        }
   else // Even phase
        {
        For j=2,4,6,8……2*(n-1)/2-1
              {
                  Apply compare-exchange(Pj, Pj+1)  //Operation is performed in parallel
              }
        }
   I++
  }
```

Let us take an example and illustrate the odd-even transposition algorithm (see Figure 11).

Analysis

The above algorithm requires one ‘for loop’ starting from I=1 to N, i.e. N times and for each value of I, one ‘for loop’ of J is executed in parallel. Therefore, the time complexity of the algorithm is O(n) as there are total n phases and each phase performs either odd or even transposition in O(1) time.

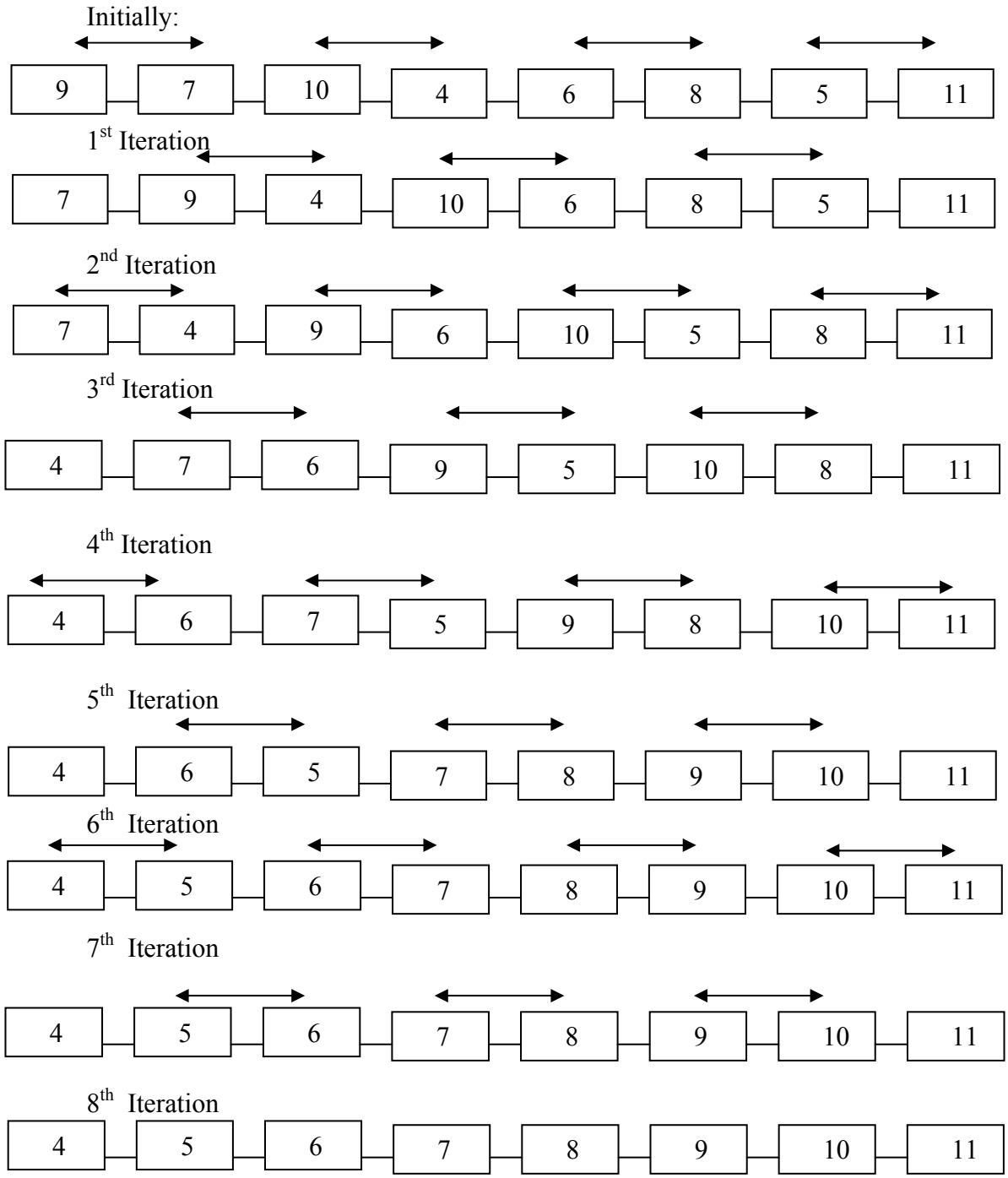


Figure 11: Example

☞ Check Your Progress 2

1) Which circuit has a time complexity of O(n) for sorting the n numbers?

1) Sort-Merge Circuit
2) Interconnection Networks
3) Combinational Circuits
4) None of the above

2) Which circuit has a time complexity of $O(\log n^2)$ for sorting the n numbers?

1) PRAM
2) Interconnection Networks
3) Combinational Circuits
4) Both 1 and 3.

3) Explain the concept of sorting in the combinational circuits

……
……
……
……

1.10 MATRIX COMPUTATION

In the subsequent section, we shall discuss the algorithms for solving the matrix multiplication problem using the parallel models.

Matrix Multiplication Problem

Let there be two matrices, M1 and M2 of sizes a x b and b x c respectively. The product of M1 x M2 is a matrix O of size a x c.
The values of elements stored in the matrix O are according to the following formulae:

$$O_{ij} = \text{Summation x of } (M1_{ix} * M2_{xj}) \text{ x=1 to b, where } 1<i<a \text{ and } 1<j<c.$$

Remember, for multiplying a matrix M1 with another matrix M2, the number of columns in M1 must equal number of rows in M2. The well known matrix multiplication algorithm on sequential computers takes $O(n^3)$ running time. For multiplication of two 2x2, matrices, the algorithm requires 8 multiplication operations and 4 addition operations. Another algorithm called Strassen algorithm has been devised which requires 7 multiplication operations and 14 addition and subtraction operations. The time complexity of Strassen's algorithm is $O(n^{2.81})$. The basic sequential algorithm is discussed below:

Algorithm: Matrix Multiplication

Input// Two Matrices M1 and M2

```
For I=1 to n
    For j=1 to n
     {
          Oij = 0;
          For k=1 to n
              Oij= Oij + M1ik * M2kj
          End For
     }
    End For
End For
```

Now, let us modify the above discussed matrix multiplication algorithm according to parallel computation models.

1.11 CONCURRENTLY READ CONCURRENTLY WRITE (CRCW)

It is one of the models based on PRAM. In this model, the processors access the memory locations concurrently for reading as well as writing operations. In the algorithm, which uses CRCW model of computation, n^3 number of processors are employed. Whenever a concurrent write operation is performed on a specific memory location, say m, than there are chances of occurrence of a conflict. Therefore, the write conflicts i.e. (WR, RW, WW) have been resolved in the following manner. In a situation when more than one processor tries to write on the same memory location, the value stored in the memory location is always the sum of the values computed by the various processors.

Algorithm Matrix Multiplication using CRCW

```
Input// Two Matrices M1 and M2
For I=1 to n          //Operation performed in PARALLEL
    For j=1 to n      //Operation performed in PARALLEL
      For k=1 to n  //Operation performed in PARALLEL
          Oij = 0;
          Oij = M1ik * M2kj
          End For
     End For
End For
```

The complexity of CRCW based algorithm is O(1).

1.12 CONCURRENTLY READ EXCLUSIVELY WRITE (CREW)

It is one of the models based on PRAM. In this model, the processors access the memory location concurrently for reading while exclusively for writing operations. In the algorithm which uses CREW model of computation, n^2 number of processors have been attached in the form of a two dimensional array of size n x n.

Algorithm Matrix Multiplication using CREW

Input// Two Matrices M1 and M2

```
For I=1 to n          //Operation performed in PARALLEL
    For j=1 to n      //Operation performed in PARALLEL
     {
          Oij = 0;
          For k=1 to n
              Oij = Oij + M1ik * M2kj
          End For
    }
    End For
End For
```

The complexity of CREW based algorithm is O(n).

☞ **Check Your Progress 3**

1) Which of the models has a complexity of O(n) for matrix multiplication?
   1) RAM
   2) Interconnection Networks
   3) CRCW
   4) CREW
2) Which of the models has a complexity of O(1) for matrix multiplication?
   1) RAM
   2) Interconnection Networks
   3) CRCW
   4) CREW
3) Explain the algorithm for matrix multiplication in sequential circuits.

……
……
……
…… …… ..

1.13 SUMMARY

In this chapter, we have discussed the various topics pertaining to the art of writing parallel algorithms for various parallel computation models in order to improve the efficiency of a number of numerical as well as non-numerical problem types. In order to evaluate the complexity of parallel algorithms there are mainly three important parameters, which are involved i.e., 1) Time Complexity, 2) Total Number of Processors Required, and 3) Total Cost Involved. Consequently, we have discussed the various computation models for parallel computers, e.g. combinational circuits, interconnection networks, PRAM etc. A combinational circuit can be defined as an arrangement of logic gates with a set of m input lines and a set of n output lines. In the interconnection networks, the N processors can communicate with each other through direct links. In the interconnection networks, each processor has an independent local memory. In the PRAM, it contains n processors, a single shared memory module being shared by all the N processors and which also acts as a communication medium for the processors. In order to connect the N processors with the single shared memory, a component called Memory Access Unit (MAU) is used for accessing the shared memory. Subsequently, we have discussed and applied these models on few numerical problems like sorting and matrix multiplication. In case of sorting, initially a combinational circuit was used for sorting. A bitonic sequence was given as an input to a combinational circuit consisting of a set of comparators interconnected with each other. The complexity of sorting using Combinational Circuit is O ($\log^2 n$). Another famous sorting algorithm known as merge sort based algorithm can also be depicted / solved with the help of the combinational circuit. The complexity of merge-sort using Combinational Circuit is O ($\log^2 n$). The interconnection network can be used for solving the sorting problem known as bubble sort. The interconnection network can be depicted as n processors interconnected with each other in the form of a linear array. The complexity of bubble sort using interconnection network is O (n). The well known matrix multiplication algorithm on sequential computers take O (n^3) running time and strassen algorithm take O($n^{2.81}$). In the present study, we have discussed two models based on PRAM for solving the matrix multiplication problem. In CRCW model, the processors access the memory location concurrently for reading as well as for writing operation. In the algorithm which uses CRCW model of computation, n^3 number of processors are employed. The complexity of CRCW based algorithm is O(1). In CREW model, the processors access the memory

location concurrently for reading while exclusively for writing operation. In the algorithm which uses CREW model of computation, n^2 number of processors have been attached in the form of a two dimensional array of size n x n. The complexity of CREW based algorithm is O(n).

1.14 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1: 1
2: 3
3: 3
4: The fundamental parameters required for the analysis of parallel algorithms are as under:

1. Time Complexity;
2. The Total Number of Processors Required; and
3. The Cost Involved.

☞ **Check Your Progress 2**

1: 2
2: 4
3: Each input line of the combinational circuit represents an individual element of the string say, X_i, and each output line results in the form of a sorted list. In order to achieve the above-mentioned task, a comparator is employed for processing.

☞ **Check Your Progress 3**

1: 4
2: 3
3: The values of elements stored in matrix O are according to the following formulae:

O_{ij} = Summation of ($M1_{ix}$ * $M2_{xj}$) x=1 to b, where 1<i<a and 1<j<c

1.15 REFERENCES/FURTHER READINGS

1) Cormen T. H., *Introduction to Algorithms*, Second Edition, Prentice Hall of India, 2002.

2) Rajaraman V. and Siva Ram Murthy C. *Parallel Computers - Architecture and Programming*, Second Edition, Prentice Hall of India, 2002.

3) Xavier C. and Iyengar S. S. *Introduction to Parallel Algorithm.*

UNIT 2 PRAM ALGORITHMS

Structure **Page Nos.**

2.0 INTRODUCTION

PRAM (Parallel Random Access Machine) model is one of the most popular models for designing parallel algorithms. A PRAM consists of unbounded number of processors interacting with each other through shared memory and a common communication network. There are many ways to implement the PRAM model. We shall discuss three of them in this unit: message passing, shared memory and data parallel. We shall also cover these models and associated data structures here.

A number of languages and routine libraries have been invented to support these models. Some of them are architecture independent and some are specific to particular platforms. We shall introduce two of the widely accepted routine libraries in this unit. These are Message Passing Interface (MPI) and Parallel Virtual Machine (PVM).

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the concepts of message passing programming;
- list the various communication modes for communication among processors;
- explain the concepts of shared programming model;
- describe and use the functions defined in MPI;
- understand and use functions defined in PVM;
- explain the concepts of data parallel programming, and
- learn about different data structures like array, linked list and hypercube.

2.2 MESSAGE PASSING PROGRAMMING

Message passing is probably the most widely used parallel programming paradigm today. It is the most natural, portable and efficient approach for distributed memory systems. It provides natural synchronisation among the processes so that explicit synchronisation of memory access is redundant. The programmer is responsible for determining all parallelism. In this programming model, multiple processes across the arbitrary number

of machines, each with its own local memory, exchange data through *send and receive* communication between processes. This model can be best understood through the diagram shown in *Figure 1*:

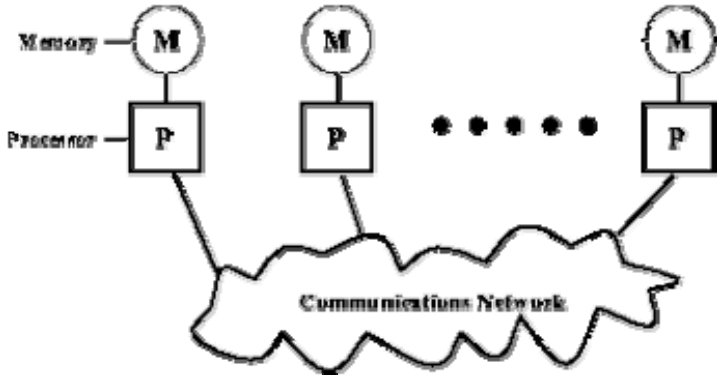


Figure 1: Message passage model

As the diagram indicates, each processor has its own local memory. Processors perform computations with the data in their own memories and interact with the other processors, as and when required, through communication network using message-passing libraries. The message contains the data being sent. But data is not the only constituent of the message. The other components in the message are:

- The identity/address of the processor that sending the message;
- Starting address of the data on the sending processor;
- The type of data being sent;
- The size of data;
- The identity/address of processor(s) are receiving the message, and
- Starting address of storage for the data on the receiving processor.

Once the message has been created it is sent through the communication network. The communication may be in the following two forms:

i) Point-to-point Communication

The simplest form of message is a point-to- point communication. A message is sent from the sending processor to a receiving processor. Message passing in point-to-point communication can be in two modes: synchronous and asynchronous. In synchronous transfer mode, the next message is sent only after the acknowledgement of delivery of the last message. In this mode the sequence of the messages is maintained. In asynchronous transfer mode, no acknowledgement for delivery is required.

ii) Collective Communications

Some message-passing systems allow communication involving more than two processors. Such type of communication may be called collective communication. Collective communication can be in one of these modes:

Barrier: In this mode no actual transfer of data takes place unless all the processors involved in the communication execute a particular block, called barrier block, in their message passing program.

Broadcast: Broadcasting may be one-to-all or all-to-all. In one-to-all broadcasting, one processor sends the same message to several destinations with a single operation whereas

in all-to-all broadcasting, communication takes place in many-to-many fashion. The messages may be personalised or non-personalized. In a personalized broadcasting, unique messages are being sent to every destination processor.

Reduction: In reductions, one member of the group collects data from the other members, reduces them to a single data item which is usually made available to all of the participating processors.

Merits of Message Passage Programming

- Provides excellent low-level control of parallelism;
- Portable;
- Minimal overhead in parallel synchronisation and data distribution; and
- It is less error prone.

Drawbacks

- Message-passing code generally requires more software overhead than parallel shared-memory code.

2.1.1 Shared Memory

In shared memory approach, more focus is on the control parallelism instead of data parallelism. In this model, multiple processes run independently on different processors, but they share a common address space accessible to all as shown in *Figure 2*.

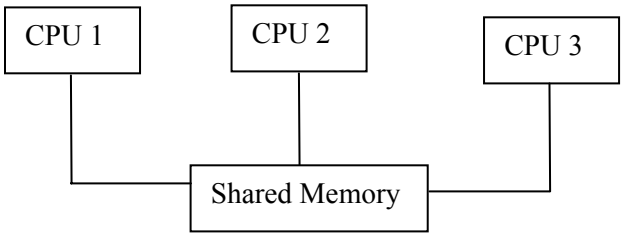


Figure 2: Shared memory

The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. Any change in a memory location effected by one processor is visible to all other processors. As shared data can be accessed by more than one processes at the same time, some control mechanism such as locks/ semaphores should be devised to ensure mutual exclusion. This model is often referred to as SMP (Symmetric Multi Processors), named so because a common symmetric implementation is used for several processors of the same type to access the same shared memory. A number of multi-processor systems implement a shared-memory programming model; examples include: NEC SX-5, SGI Power Onyx/ Origin 2000; Hewlett-Packard V2600/HyperPlex; SUN HPC 10000 400 MHz ;DELL PowerEdge 8450.

Shared memory programming has been implemented in various ways. We are introducing some of them here.

Thread libraries

The most typical representatives of shared memory programming models are thread libraries present in most of the modern operating systems. Examples for thread libraries

are, *POSIX threads* as implemented in Linux, *SolarisTM threads* for solaris , *Win32 threads* available in Windows NT and Windows 2000 , and *JavaTM threads* as part of the standard JavaTM Development Kit (JDK).

Distributed Shared Memory Systems

Distributed Shared Memory (DSM) systems emulate a shared memory abstraction on loosely coupled architectures in order to enable shared memory programming despite missing hardware support. They are mostly implemented in the form of standard libraries and exploit the advanced user–level memory management features present in modern operating systems. Examples include Tread Marks System, IVY, Munin, Brazos, Shasta, and Cashmere.

Program Annotation Packages

A quite renowned approach in this area is OpenMP, a newly developed industry standard for shared memory programming on architectures with uniform memory access characteristics. OpenMP is based on functional parallelism and focuses mostly on the parallelisation of loops. OpenMP implementations use a special compiler to evaluate the annotations in the application's source code and to transform the code into an explicitly parallel code, which can then be executed. We shall have a detailed discussion on OpenMP in the next unit.

Shared memory approach provides low-level control of shared memory system, but they tend to be tedious and error prone. They are more suitable for system programming than to application programming.

Merits of Shared Memory Programming

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between processes is both fast and uniform due to the proximity of memory to CPUs.
- No need to specify explicitly the communication of data between processes.
- Negligible process-communication overhead.
- More intuitive and easier to learn.

Drawbacks

- Not portable.
- Difficult to manage data locality.
- Scalability is limited by the number of access pathways to memory.
- User is responsible for specifying synchronization, e.g., locks.

2.1.2 Message Passing Libraries

In this section, we shall discuss about message passing libraries. Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications. We shall discuss only two worldwide accepted message passing libraries namely; MPI and PVM.

Message Passing Interface (MPI)

The **Message Passing Interface** (**MPI**) is a universal standard for providing communication among the multiple concurrent processes on a distributed memory system. Most, if not all, of the popular parallel computing platforms offer at least one implementation of MPI. It was developed by the MPI forum consisting of several experts

from industry and academics. MPI has been implemented as the library of routines that can be called from languages like, Fortran, C, C++ and Ada programs. MPI was developed in two stages, MPI-1 and MPI-2. MPI-1 was published in 1994.

Features of MPI-1

- Point-to-point communication,
- Collective communication,
- Process groups and communication domains,
- Virtual process topologies, and
- Binding for Fortran and C.

Features added in MPI-2

- Dynamic process management,
- Input/output,
- One-sided operations for remote memory access, and
- Binding for C++.

MPI's advantage over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware it runs on).

Building and Running MPI Programs

MPI parallel programs are written using conventional languages like, Fortran and C .One or more header files such as "mpi.h" may be required to provide the necessary definitions and declarations. Like any other serial program, programs using MPI need to be compiled first before running the program. The command to compile the program may vary according to the compiler being used. If we are using *mpcc* compiler, then we can compile a C program named "program.c" using the following command:

mpcc program.c -o program.o

Most implementations provide command, typically named *mpirun* for spawning MPI processes. It provides facilities for the user to select number of processes and which processors they will run on. For example to run the object file "program" as n processes on n processors we use the following command:

mpirun program -np n

MPI functions

MPI includes hundreds of functions, a small subset of which is sufficient for most practical purposes. We shall discuss some of them in this unit.

Functions for MPI Environment:

int MPI_Init (int *argc, char ** argv)

It initializes the MPI environment. No MPI function can be called before MPI_Init.

int MPI_Finalize (void)

It terminates the MPI environment. No MPI function can be called after MPI_Finalize.

Every MPI process belongs to one or more groups (also called communicator). Each process is identified by its rank (0 to group size –1) within the given group. Initially, all

processes belong to a default group called MPI_COMM_WORLD group. Additional groups can be created by the user as and when required. Now, we shall learn some functions related to communicators.

int MPI_Comm_size (MPI_Comm comm, int *size)

returns variable *size* that contains number of processes in group *comm*.

int MPI_Comm_rank (MPI_Comm comm, int *rank)

returns **rank** of calling process in group *comm*.

Functions for Message Passing:

MPI processes do not share memory space and one process cannot directly access other process's variables. Hence, they need some form of communication among themselves. In MPI environment this communication is in the form of message passing. A message in MPI contains the following fields:

msgaddr: It can be any address in the sender's address space and refers to location in memory where message data begins.

count: Number of occurrences of data items of message datatype contained in message.

datatype: Type of data in message. This field is important in the sense that MPI supports heterogeneous computing and the different nodes may interpret *count* field differently. For example, if the message contains a strings of 2n characters (count =2n), some machines may interpret it having 2n characters and some having n characters depending upon the storage allocated per character (1 or 2). The basic datatype in MPI include all basic types in Fortran and C with two additional types namely MPI_BYTE and MPI_PACKED. MPI_BYTE indicates a byte of 8 bits .

source or *dest* : Rank of sending or receiving process in communicator.

tag: Identifier for specific message or type of message. It allows the programmer to deal with the arrival of message in an orderly way, even if the arrival of the message is not orderly.

comm.: Communicator. It is an object wrapping context and group of a process .It is allocated by system instead of user.

The functions used for messaging passing are:

int MPI_Send(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* can be reused immediately.

int MPI_Recv(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* contains requested message.

MPI message passing may be either point-to-point or collective.

Point-to-point Message Passing

In point-to-point message passing, one process sends/receives message to/from another process. There are four communication modes for sending a message:

i) **Buffered mode:** Send can be initiated whether or not matching receive has been initiated, and send may complete before matching receive is initiated.

ii) **Synchronous mode:** Send can be initiated whether or not matching receive has been initiated, but send will complete only after matching receive has been initiated.

iii) **Ready mode:** Send can be initiated only if matching receive has already been initiated.

iv) **Standard mode:** May behave like either buffered mode or synchronous mode, depending on specific implementation of MPI and availability of memory for buffer space.

MPI provides both blocking and non-blocking send and receive operations for all modes.

Functions for various communication modes

Mode	Blocking	Non-Blocking
Standard	*MPI_Send*	*MPI_Isend*
Buffered	*MPI_Bsend*	*MPI_Ibsend*
Synchronus	*MPI_Ssend*	*MPI_Issend*
Ready	*MPI_Rsend*	*MPI_Irsend*

MPI_Recv and *MPI_Irecv* are blocking and nonblocking functions for receiving messages, regardless of mode.

Besides send and receive functions, MPI provides some more useful functions for communications. Some of them are being introduced here.

MPI_Buffer_attach used to provide buffer space for buffered mode. Nonblocking functions include *request* argument used subsequently to determine whether requested operation has completed.

MPI_Wait and *MPI_Test* wait or test for completion of nonblocking communication.

MPI_Probe and *MPI_Iprobe* probe for incoming message without actually receiving it. Information about message determined by probing can be used to decide how to receive it.

MPI_Cancel cancels outstanding message request, useful for cleanup at the end of a program or after major phase of computation.

Collective Message Passing

In collective message passing, all the processes of a group participate in communication. MPI provides a number of functions to implement the collective message passing. Some of them are being discussed here.

MPI_Bcast(msgaddr, count, datatype, rank, comm):

This function is used by a process ranked *rank* in group *comm* to broadcast the message to all the members (including self) in the group.

MPI_Allreduce

***MPI_Scatter*(Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype, Rank, Comm):**

Using this function process with rank rank *in group comm sends personalized message to all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the send buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.*

MPI_Gather (Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype,Rank, Comm):
Using this function process with rank rank in group comm receives personalized message from all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the receive buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.

MPI_Alltoall()
Each process sends a personalized message to every other process in the group.

MPI_Reduce (Sendaddr , Receiveaddr , count, datatype, op, rank, comm):

This function reduces the partial values stored in Sendaddr of each process into a final result and stores it in Receiveaddr of the process with rank rank. op specifies the reduction operator**.**

***MPI_Scan** (Sendaddr,, Receiveaddr , count, datatype, op, comm):*
It combines the partial values into p final results which are received into the Receiveaddr of all p processes in the group comm.

MPI_Barrier(cmm):
This function synchronises all processes in the group comm.

Timing in MPI program

MPI_Wtime ( ) returns elapsed wall-clock time in seconds since some arbitrary point in past. Elapsed time for program segment is given by the difference between *MPI_Wtime* values at beginning and end of process. Process clocks are not necessarily synchronised, so clock values are not necessarily comparable across the processes, and care must be taken in determining overall running time for parallel program. Even if clocks are explicitly synchronised, variation across clocks still cannot be expected to be significantly less than round-trip time for zero-length message between the processes.

Now, we shall illustrate use of these functions with an example.

Example 1:

```
#include <mpi.h>
int main(int argc, char **argv) {
int i, tmp, sum, s, r, N, x[100];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
If(r==0)
 {
  printf( “Enter N:”);
  scanf (“%d”, &N);
  for (i=1; i<s; i++)
     MPI_Send(&N, 1, MPI_INT,i, i, MPI_COMM_WORLD);
  for (i=r, i<N; i=i+s)
     sum+= x[i];
```

```
 for (i=r, i<s; i++)
 {
   MPI_Recv(&tmp, 1, MPI_INT,i, i,  MPI_COMM_WORLD, &status);
   Sum+=tmp;
}
printf( "%d", sum);
}
else {
        MPI_Recv(&N, 1, MPI_INT,0, i, MPI_COMM_WORLD, &status);
        for (i=r, i<N; i=i+s)
             sum+= x[i];
        MPI_Send(&sum, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
}
MPI_Finalize( );
}
```

Merits of MPI

- Runs on either shared or distributed memory architectures;
- Can be used on a wider range of problems than OpenMP;
- Each process has its own local variables; and
- Distributed memory computers are less expensive than large shared memory computers.

Drawbacks of MPI

- Requires more programming changes to go from serial to parallel version.
- Can be harder to debug, and
- Performance is limited by the communication network between the nodes.

Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a portable message-passing programming system, designed to link separate heterogeneous host machines to form a "virtual machine" which is a single, manageable parallel computing resource. Large computational problems such as molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, can thus be solved more cost effectively by using the aggregate power and memory of many computers.

PVM was developed by the University of Tennessee, The Oak Ridge National Laboratory and Emory University. The first version was released in 1989, version 2 was released in 1991 and finally version 3 was released in 1993. The PVM software enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. It transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures. The programming interface of PVM is very simple .The user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronisation between the tasks. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast and barrier synchronization.

Features of PVM:

- Easy to install;

- Easy to configure;
- Multiple users can each use PVM simultaneously;
- Multiple applications from one user can execute;
- C, C++, and Fortran supported;
- Package is small;
- Users can select the set of machines for a given run of a PVM program;
- Process-based computation;
- Explicit message-passing model, and
- Heterogeneity support.

When the PVM is starts it examines the virtual machine in which it is to operate, and creates a process called the PVM demon, or simply pvmd on each machine. An example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing electronic mail on a computer, pvmd provides inter-host point of contact, authenticates task and executes processes on machines. It also provides the fault detection, routes messages not from or intended for its host, transmits messages from its application to a destination, receives messages from the other pvmd's, and buffers it until the destination application can handle it.

PVM provides a library of functions, libpvm3.a, that the application programmer calls. Each function has some particular effect in the PVM. However, all this library really provides is a convenient way of asking the local pvmd to perform some work. The pvmd then acts as the virtual machine. Both pvmd and PVM library constitute the PVM system.

The PVM system supports functional as well as data decomposition model of parallel programming. It binds with C, C++, and Fortran . The C and C++ language bindings for the PVM user interface library are implemented as functions (subroutines in case of FORTRAN) . User programs written in C and C++ can access PVM library by linking the library libpvm3.a (libfpvm3.a in case of FORTRAN).

All PVM tasks are uniquely identified by an integer called *task identifier* (TID) assigned by local pvmd. Messages are sent to and received from tids. PVM contains several routines that return TID values so that the user application can identify other tasks in the system. PVM also supports grouping of tasks. A task may belong to more than one group and one task from one group can communicate with the task in the other groups. To use any of the group functions, a program must be linked with `libgpvm3.a.`

Set up to Use PVM

PVM uses two environment variables when starting and running. Each PVM user needs to set these two variables to use PVM. The first variable is PVM_ROOT, which is set to the location of the installed `pvm3` directory. The second variable is PVM_ARCH , which tells PVM the architecture of this host. The easiest method is to set these two variables in your `.`cshrc file. Here is an example for setting PVM_ROOT:

```
setenv PVM_ROOT $HOME/pvm3
```

The user can set PVM_ARCH by concatenating to the file `.cshrc`, the content of file `$PVM_ROOT/lib/cshrc.stub`.

Starting PVM

To start PVM, on any host on which PVM has been installed we can type

```
% pvm
```

The PVM console, called `pvm`, is a stand-alone PVM task that allows the user to interactively start, query, and modify the virtual machine. Then we can add hosts to virtual machine by typing at the console prompt (got after last command)

```
pvm> add hostname
```

To delete hosts (except the one we are using) from virtual machine we can type

```
pvm> delete hostname
```

We can see the configuration of the present virtual machine, we can type

```
pvm> conf
```

To see what PVM tasks are running on the virtual machine, we should type

```
pvm> ps -a
```

To close the virtual machine environment, we should type

```
pvm> halt
```

Multiple hosts can be added simultaneously by typing the hostnames in a file one per line and then type

```
% pvm hostfile
```

PVM will then add all the listed hosts simultaneously before the console prompt appears.

Compiling and Running the PVM Program

Now, we shall learn how to compile and run PVM programs. To compile the program , change to the directory pvm/lib/archname where archname is the architecture name of your computer. Then the following command:

```
cc  program.c -lpvm3 -o prgram
```

will compile a program called program.c. After compiling, we must put the executable file in the directory pvm3/bin/ARCH. Also, we need to compile the program separately for every architecture in virtual machine. In case we use dynamic groups, we should also add -lgpvm3 to the compile command. The executable file can then be run. To do this, first run PVM. After PVM is running, executable file may be run from the unix command line, like any other program.

PVM supplies an architecture-independent make, `aimk`, that automatically determines PVM_ARCH and links any operating system specific libraries to your application. To compile the C example, type

```
% aimk master.c
```

Now, from one window, start PVM and configure some hosts. In another window change directory to `$HOME/pvm3/bin/PVM_ARCH` and type

```
% master
```

It will ask for a number of tasks to be executed. Then type the number of tasks.

Programming with PVM

The general method for writing a program with PVM is as follows:
A user writes one or more sequential programs in C, C++, or Fortran 77 containing embedded PVM function (or subroutine) calls. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem.

PVM library routines

In this section we shall give a brief description of the routines in the PVM 3 user library. Every PVM program should include the PVM header file "`pvm3.h`" (in a C program) or "fpvm3.h" (in a Fortran program).

In PVM 3, all PVM tasks are identified by an integer supplied by the local pvmd. In the following descriptions this task identifier is called TID. Now we are introducing some commonly used functions in PVM programming (as in C. For Fortran, we use prefix pvmf against pvm in C).

Process Management

- int **pvm_mytid(void)**

Returns the *tid* of the calling process. **tid** values less than zero indicate an error.

- **int pvm_exit(void)**

Tells the local pvmd that this process is leaving PVM. **info** Integer status code returned by the routine. Values less than zero indicate an error.

- **pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)**

start new PVM processes. **task**, a character string is the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. **Argv is a** pointer to an array of arguments to **task**. If the executable needs no arguments, then the second argument to pvm_spawn is NULL. **flag** Integer specifies spawn options. **where** , a character string specifying where to start the PVM process. If *flag* is 0, then *where* is ignored and PVM will select the most appropriate host. **ntask** ,an integer, specifies the number of copies of the executable to start. **tids** ,Integer array of length *ntask* returns the tids of the PVM processes started by this pvm_spawn call. The function returns the actual number of processes returned. Negative values indicate error.

- **int pvm_kill(int tid)**

Terminates a specified PVM process. **tid** Integer task identifier of the PVM process to be killed (not itself). Return values less than zero indicate an error.

- **int pvm_catchout(FILE *ff)**

Catch output from child tasks. **ff** is file descriptor on which we write the collected output. The default is to have the PVM write the *stderr* and *stdout* of spawned tasks.

Information

- **int pvm_parent(void)**

Returns the tid of the process that spawned the calling process.

- **int pvm_tidtohost(tid)**

Returns the host of the specified PVM process. Error if negative value is returned.

- **int pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)**

```
struct pvmhostinfo {
int hi_tid;
char *hi_name;
char *hi_arch;
int hi_speed;
};
```

Returns information about the present virtual machine configuration. **nhost** is the number of hosts (pvmds) in the virtual machine. **narch** is the number of different data formats being used. **hostp** is pointer to an array of structures which contains the information about each host including its pvmd task ID, name, architecture, and relative speed(default is 1000).

- **int info = pvm_tasks(int where, int *ntask, struct pvmtaskinfo **taskp)**

```
struct pvmtaskinfo {
    int ti_tid; int ti_ptid;
    int ti_host;
    int ti_flag; char *ti_a_out; } taskp;
```

Returns the information about the tasks running on the virtual machine. **where** specifies what tasks to return the information about. The options are:

0
    for all the tasks on the virtual machine

pvmd tid
    for all tasks on a given host

tid
    for a specific task

ntask returns the number of tasks being reported on.

taskp is a pointer to an array of structures which contains the information about each task including its task ID, parent tid, pvmd task ID, status flag, and the name of this task's executable file. The status flag values are: waiting for a message, waiting for the pvmd, and running.

Dynamic Configuration

- **int pvm_addhosts(char **hosts, int nhost, int *infos)**
  Add hosts to the virtual machine. **hosts** is an array of strings naming the hosts to be added. **nhost** specifies the length of array hosts. **infos** is an array of length *nhost* which returns the status for each host. Values less than zero indicate an error, while positive values are TIDs of the new hosts.
  Signaling

- **int pvm_sendsig(int tid, int signum**)

Sends a signal to another PVM process. **tid** is task identifier of PVM process to receive the signal. **signum** is the signal number.

- **int info = pvm_notify(int what, int msgtag, int cnt, int *tids)**

Request notification of PVM event such as host failure. **What** specifies the type of event to trigger the notification. Some of them are:

PvmTaskExit
: Task exits or is killed.

PvmHostDelete
: Host is deleted or crashes.

PvmHostAdd
: New host is added.

msgtag is message tag to be used in notification. **cnt** For *PvmTaskExit* and *PvmHostDelete*, specifies the length of the *tids* array. For *PvmHostAdd* specifies the number of times to notify.

tids for *PvmTaskExit* and *PvmHostDelete* is an array of length *cnt* of task or pvmd TIDs to be notified about. The array is not used with the *PvmHostAdd* option.

Message Passing

The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse, and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. There are options in PVM 3 that request that data be transferred directly from task to task. In this case, if the message is large, the sender may block until the receiver has called a matching receive.

A nonblocking receive immediately returns with either the data or a flag that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer. In addition to these point-to-point communication functions, the model supports the multicast to a set of tasks and the broadcast to a user-defined group of tasks. There are also functions to perform global max, global sum, etc. across a user-defined group of tasks. Wildcards can be specified in the receive for the source and the label, allowing either or both of these contexts to be ignored. A routine can be called to return the information about the received messages.

The PVM model guarantees that the message order is preserved. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both the messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

- **int bufid = pvm_mkbuf(int encoding)**

creates a new message buffer. **encoding** specifying the buffer's encoding scheme.

Encoding data options

PVM uses SUN's XDR library to create a machine independent data format if you request it. Settings for the encoding option are:

PvmDataDefault: Use XDR by default, as the local library cannot know in advance where you are going to send the data.

PvmDataRaw: No encoding, so make sure you are sending to a similar machine.

PvmDataInPlace: Not only is there no encoding, but the data is not even going to be physically copied into the buffer.

- **int info = pvm_freebuf(int bufid)**

Disposes of a message buffer. **bufid** message buffer identifier.

- **int pvm_getsbuf(void)**

returns the active send buffer identifier.

- **int pvm_getrbuf(void)**

returns the active receive buffer identifier.

- **int pvm_setsbuf(int bufid)**

sets the active send buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **int pvm_setrbuf(int bufid)**

sets the active receive buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **intpvm_initsend(int encoding)**

Clear default sends buffer and specifies the message encoding. **Encoding** specifies the next message's encoding scheme.

- **int pvm_send(int tid, int msgtag)**

Immediately sends the data in the active message buffer. **tid** Integer task identifier of destination process. **msgtag** Integer message tag supplied by the user. msgtag should be nonnegative.

- **int pvm_recv(int tid, int msgtag)**

Receive a message. **tid** is integer task identifier of sending process supplied by the user and **msgtag** is the message tag supplied by the user(should be non negative integer). The process returns the value of the new active receive buffer identifier. Values less than zero indicate an error. It blocks the process until a message with label *msgtag* has arrived from *tid*. pvm_recv then places the message in a new *active* receive buffer, which also clears the current receive buffer.

Packing and Unpacking Data

- pvm_packs - Pack the active message buffer with arrays of prescribed data type:

- **int info = pvm_packf(const char *fmt, ...)**
- **int info = pvm_pkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_pkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_pkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_pkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_pkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_pkint(int *ip, int nitem, int stride)**
- **int info = pvm_pkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_pkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_pkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_pklong(long *ip, int nitem, int stride)**
- **int info = pvm_pkshort(short *jp, int nitem, int stride)**
- **int info = pvm_pkstr(char *sp)**

fmt Printf-like format expression specifying what to pack. **nitem** is the total number of *items* to be packed (not the number of bytes). **stride** is the stride to be used when packing the items.

- pvm_unpack - Unpacks the active message buffer into arrays of prescribed data type. It has been implemented for different data types:

- **int info = pvm_unpackf(const char *fmt, ...)**
- **int info = pvm_upkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_upkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_upkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_upkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_upkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_upkint(int *ip, int nitem, int stride)**
- **int info = pvm_upkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_upkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_upkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_upklong(long *ip, int nitem, int stride)**
- **int info = pvm_upkshort(short *jp, int nitem, int stride)**
- **int info = pvm_upkstr(char *sp)**

Each of the pvm_upk* routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, *nitem* which is the total number of items to unpack, and *stride* which is the stride to use when unpacking. An exception is pvm_upkstr() which by definition unpacks a NULL terminated character string and thus does not need *nitem* or *stride* arguments.

Dynamic Process Groups

To create and manage dynamic groups, a separate library `libgpvm3.a` must be linked with the user programs that make use of any of the group functions. Group management work is handled by a group server that is automatically started when the first group function is invoked. Any PVM task can join or leave any group dynamically at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not members. Now we are giving some routines that handle dynamic processes:

- **int pvm_joingroup(char *group)**

Enrolls the calling process in a named group. **group** is a group name of an existing group. Returns instance number.Instance numbers run from 0 to the number of group members minus 1. In PVM 3, a task can join multiple groups. If a process leaves a group and then rejoins it, that process may receive a different instance number.

- **int info = pvm_lvgroup(char *group)**

Unenrolls the calling process from a named group.

- **int pvm_gettid(char *group, int inum)**

Returns the tid of the process identified by a group name and instance number.

- **int pvm_getinst(char *group, int tid)**

Returns the instance number in a group of a PVM process.

- **int size = pvm_gsize(char *group)**

Returns the number of members presently in the named group.

- **int pvm_barrier(char *group, int count)**

Blocks the calling process until all the processes in a group have called it. **count** species the number of group members that must call pvm_barrier before they are all released.

- **int pvm_bcast(char *group, int msgtag)**

Broadcasts the data in the active message buffer to a group of processes. **msgtag** is a message tag supplied by the user. It allows the user's program to distinguish between different kinds of messages .It should be a nonnegative integer.

- **int info = pvm_reduce(void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootginst)**

Performs a reduce operation over members of the specified group. **func** is function defining the operation performed on the global data. Predefined are PvmMax, PvmMin, PvmSum and PvmProduct. Users can define their own function. **data** is pointer to the starting address of an array of local values. **count** species the number of elements of *datatype* in the data array. **Datatype** is the type of the entries in the data array. **msgtag** is the message tag supplied by the user. msgtag should be greater than zero. It allows the user's program to distinguish between different kinds of messages. **group** is the group name of an existing group. **rootginst** is the instance number of group member who gets the result.

We are writing here a program that illustrates the use of these functions in the parallel programming:

Example 2: Hello.c

```
#include "pvm3.h"
main()
{
   int cc, tid, msgtag;
   char buf[100];

   printf("%x\n", pvm_mytid());

   cc = pvm_spawn("hello_other", (char**)0, 0, "", 1,
&tid);

   if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
   } else
      printf("can't start hello_other\n");

   pvm_exit();
}
```

In this program, pvm_mytid( ), returns the TID of the running program (In this case, task id of the program hello.c). This program is intended to be invoked manually; after

printing its task id (obtained with `pvm_mytid()`), it initiates a copy of another program called *hello_other* using the `pvm_spawn()` function. A successful spawn causes the program to execute a blocking receive using `pvm_recv`. After receiving the message, the program prints the message sent by its counterpart, as well its task id; the buffer is extracted from the message using `pvm_upkstr`. The final `pvm_exit` call dissociates the program from the PVM system.

hello_other.c

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the ``slave'' or spawned program; its first PVM action is to obtain the task id of the ``master'' using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

☞ **Check Your Progress 1**

1) Write a program to give a listing of the “slave” or spawned program.

……
……
……
……
……

2.2.3 Data Parallel Programming

In the data parallel programming model, major focus is on performing simultaneous operations on a data set. The data set is typically organized into a common structure, such as an array or hypercube. Processors work collectively on the same data structure. However, each task works on a different partition of the same data structure. It is *more restrictive* because not all algorithms can be specified in the data-parallel terms. For these reasons, data parallelism, although important, is not a universal parallel programming paradigm.

Programming with the data parallel model is usually accomplished by writing a program with the data parallel constructs. The constructs can be called to a data parallel subroutine

library or compiler directives. Data parallel languages provide facilities to specify the data decomposition and mapping to the processors. The languages include data distribution statements, which allow the programmer to control which data goes on what processor to minimize the amount of communication between the processors. Directives indicate how arrays are to be aligned and distributed over the processors and hence specify agglomeration and mapping. Communication operations are not specified explicitly by the programmer, but are instead inferred by the compiler from the program. Data parallel languages are more suitable for SIMD architecture though some languages for MIMD structure have also been implemented. Data parallel approach is more effective for highly regular problems, but are not very effective for irregular problems.

The main languages used for this are Fortran 90, High Performance Fortran (HPF) and HPC++. We shall discuss HPF in detail in the next unit. Now, we shall give a brief overview of some of the early data parallel languages:

- ***Computational Fluid Dynamics:*** CFD was a FORTRAN-like language developed in the early 70s at the Computational Fluid Dynamics Branch of Ames Research Center for ILLIAC IV machines, a SIMD computer for array processing. The language design was extremely pragmatic. No attempt was made to hide the hardware peculiarities from the user; in fact, every attempt was made to give the programmers the access and control of all of the hardware to help constructing efficient programs. This language made the architectural features of the ILLIAC IV very apparent to the programmer, but it also contained the seeds of some practical programming language abstractions for data-parallel programming. In spite of its simplicity and ad hoc machine-dependencies, CFD allowed the researchers at Ames to develop a range of application programs that efficiently used the ILLIAC IV.

- ***Connection Machine Fortran:*** Connection Machine Fortran was a later SIMD language developed by Thinking Machines Corporation. Connection Machine Fortran included all of FORTRAN 77, together with the new array syntax of Fortran 90. It added various machine specific features, but unlike CFD or DAP FORTRAN these appeared as *compiler directives* rather than special syntax in Fortran declarations or executable statements. A major improvement over the previous languages was that, distributed array dimensions were no longer constrained to exactly fit in the size of the processing element array; the compiler could transparently map dimensions of arbitrary extent across the available processor grid dimensions. Finally the language added an explicitly parallel looping construct called FORALL. Although CM Fortran looked syntactically like standard Fortran, the programmer had to be aware of many nuances. Like the ILLIAC IV, the Connection

Machine allowed the Fortran arrays to either be distributed across the processing nodes (called *CM arrays*, or distributed arrays), or allocated in the memory of the front-end computer (called *front-end arrays*, or sequential arrays). Unlike the control unit of the ILLIAC, the Connection Machine front-end was a conventional, general-purpose computer--typically a VAX or Sun. But there were still significant restrictions on how arrays could be manipulated, reflecting the two possible homes.

Glypnir ,IVTRAN and *LISP are some of the other early data parallel languages.

Let us conclude this unit with the introduction of a typical data parallel programming style called **SPMD.**

Single Program Multiple Date

A common style of writing data parallel programs for MIMD computers is SPMD (single program, multiple data): all the processors execute the same program, but each operates

on a different portion of problem data. It is easier to program than true MIMD, but more flexible than SIMD. Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style. In this style, although there is no central controller, the worker nodes carry on doing *essentially* the same thing at *essentially* the same time. Instead of central copies of control variables stored on the control processor of a SIMD computer, control variables (iteration counts and so on) are usually stored in a replicated fashion across MIMD nodes. Each node has its own local copy of these global control variables, but every node updates them in an identical way. There are no centrally issued parallel instructions, but communications usually happen in the well-defined collective phases. These data exchanges occur in a prefixed manner that explicitly or implicitly synchronize the peer nodes. The situation is something like an orchestra without a conductor. There is no central control, but each individual plays from the same script. The group as a whole stays in lockstep. This loosely synchronous style has some similarities to the Bulk Synchronous Parallel (BSP) model of computing introduced by the theorist Les Valiant in the early 1990s. The restricted pattern of the collective synchronization is easier to deal with than the complex synchronisation problems of a general concurrent programming.

A natural assumption was that it should be possible and not too difficult to capture the SPMD model for programming MIMD computers in data-parallel languages, along lines similar to the successful SIMD languages. Various research prototype languages attempted to do this, with some success. By the 90s the value of portable, standarised programming languages was universally recognized, and there seemed to be some consensus about what a standard language for SPMD programming ought to look like. Then the High Performance Fortran (HPF) standard was introduced.

2.3 DATA STRUCTURES FOR PARALLEL ALGORITHMS

To implement any algorithm, selection of a proper data structure is very important. A particular operation may be performed with a data structure in a smaller time but it may require a very large time in some other data structure. For example, to access i^{th} element in a set may need constant time if we are using arrays but the required time becomes a polynomial in case of a linked list. So, the selection of data structure must be done keeping in mind the type of operation to be performed and the architecture available. In this section, we shall introduce some data structures commonly used in a parallel programming.

2.3.1 Linked List

A linked list is a data structure composed of zero or more nodes linked by pointers. Each node consists of two parts, as shown in *Figure 3*: *info* field containing specific information and *next* field containing address of next node. First node is pointed by an external pointer called *head*. Last node called tail node does not contain address of any node. Hence, its next field points to null. Linked list with zero nodes is called null linked list.

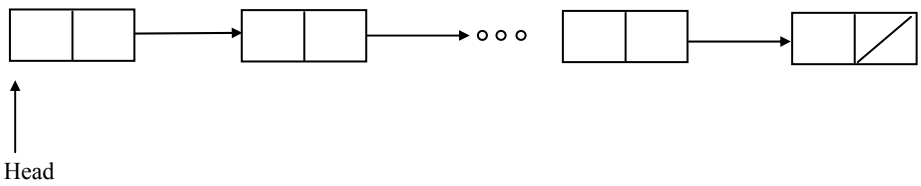


Figure 3: Linked List

A large number of operations can be performed using the linked list. For some of the operations like insertion or deletion of the new data, linked list takes constant time, but it is time consuming for some other operations like searching a data. We are giving here an example where linked list is used:

Example 3:
Given a linear linked list, rank the list elements in terms of the distance from each to the last element.

A parallel algorithm for this problem is given here. The algorithm assumes there are p number of processors.

Algorithm:

```
Processor j, 0≤ j<p, do
  if next[j]=j then
    rank[j]=0
   else rank[j] =1
 endif
 while rank[next[first]]≠0 Processor j, 0≤ j<p, do
      rank[j]=rank[j]+rank[next[j]]
      next[j]=next[next[j]]
 endwhile
```

The working of this algorithm is illustrated by the following diagram:

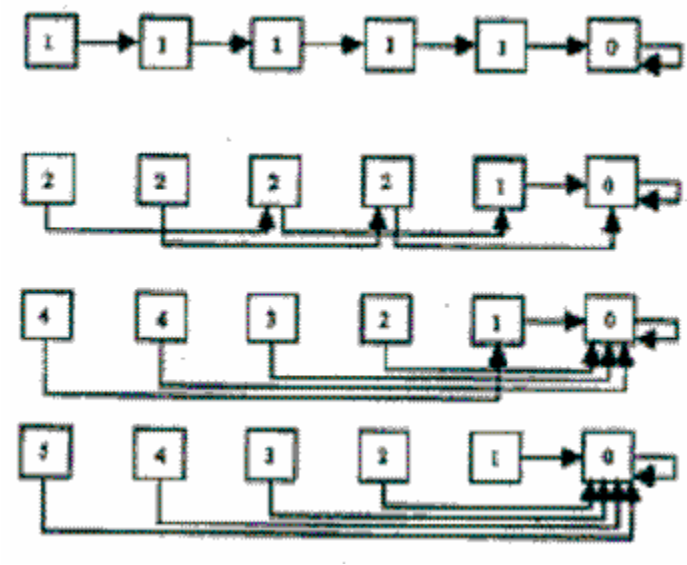


Figure 4 : Finding rank of elements

2.3.2 Arrays Pointers

An array is a collection of the similar type of data. Arrays are very popular data structures in parallel programming due to their easiness of declaration and use. At the one hand, arrays can be used as a common memory resource for the shared memory programming, on the other hand they can be easily partitioned into sub-arrays for data parallel programming. This is the flexibility of the arrays that makes them most

frequently used data structure in parallel programming. We shall study arrays in the context of two languages Fortran 90 and C.

Consider the array shown below. The size of the array is 10.

5	10	15	20	25	30	35	40	45	50

Index of the first element in Fortran 90 is 1 but that in C is 0 and consequently the index of the last element in Fortran 90 is 10 and that in C is 9. If we assign the name of array as A, then i^{th} element in Fortran 90 is A(i) but in C it is A[i-1]. Arrays may be one-dimensional or they may be multi-dimensional.

General form of declaration of array in Fortran 90 is

```
type, DIMENSION(bound) [,attr] :: name
```

for example the declaration

```
INTEGER, DIMENSION(5): A
```

declare an array A of size 5.

General form of declaration of array in C is

type array_name [size]

For example the declaration A

int A[10]

declares an array of size 10.

Fortran 90 allows one to use particular sections of an array. To access a section of an array, you need the name of the array followed by the two integer values separated by a colon enclosed in the parentheses. The integer values represent the indices of the section required.
For example, a(3:5) refers to elements 3, 4, 5 of the array, a(1:5:2) refers to elements 1, 3, 5 of the array , and b(1:3, 2:4) refers to the elements from rows 1 to 3 and columns 2 to 4. In C there is only one kind of array whose size is determined statically, though there are provisions for dynamic allocation of storage through pointers and dynamic memory allocation functions like *calloc* and *malloc* functions. In Fortran 90, there are 3 possible types of arrays depending on the binding of an array to an amount of storage : *Static arrays* with fixed size at the time of declaration and cannot be altered during execution ; *Semi-dynamic arrays* or *automatic arrays*: the size is determined after entering a subroutine and arrays can be created to match the exact size required, but local to a subroutine ; and *Dynamic arrays* or *allocatable arrays* : the size can be altered during execution.

In these languages, array operations are written in a compact form that often makes programs more readable.
Consider the loop:

```
s=0
do i=1,n
   a(i)=b(i)+c(i)
    s=s+a(i)
 end do
```

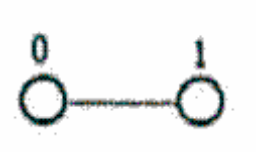
It can be written (in Fortran 90 notation) as follows:

```
a(1:n) = b(1:n) +c(1:n)
s=sum(a(1:n))
```

In addition to Fortran 90 , there are many languages that provide succinct operations on arrays. Some of the most popular are APL, and MATLAB. Although these languages were not developed for parallel computing, rather for expressiveness, they can be used to express parallelism since array operations can be easily executed in parallel. Thus, all the arithmetic operations (+, -, * /, **) involved in a vector expression can be performed in parallel. Intrinsic reduction functions, such as the sum above, also can be performed in a parallel .

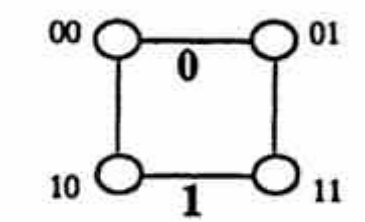
2.3.3 Hypercube Network

The hypercube architecture has played an important role in the development of parallel processing and is still quite popular and influential. The highly symmetric recursive structure of the hypercube supports a variety of elegant and efficient parallel algorithms. Hypercubes are also called n-cubes, where n indicates the number of dimensions. An-cube can be defined recursively as depicted below:



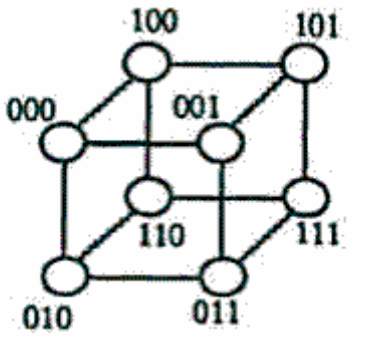
1-cube built of 2 0-cubes

Figure 5(a): 1-cube



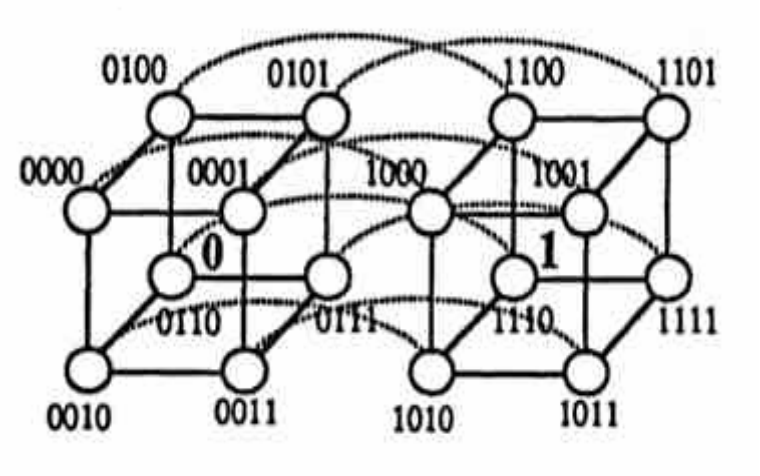
2-cube built of 2 1-cubes

Figure 5(b): 2-cube



3-cube built of 2 2-cubes

Figure 5(c): 3-cube



4-cube built of 2 3-cubes

Figure 5(d): 4-cube

Properties of Hypercube:

- A node p in a n-cube has a unique label, its binary ID, that is a n-bit binary number.
- The labels of any two neighboring nodes differ in exactly 1 bit.
- Two nodes whose labels differ in *k* bits are connected by a shortest path of length *k*.
- Hypercube is both node- and edge- symmetric.

Hypercube structure can be used to implement many parallel algorithms requiring all-to-all communication, that is, algorithms in which each task must communicate with every other task. This structure allows a computation requiring all-to-all communication among P tasks to be performed in just log P steps compared t polynomial time using other data structures like arrays and linked lists.

2.4 SUMMARY

In this unit, a number of concepts have been introduced in context designing of algorithms for PRAM model of parallel computation. The concepts introduced include message passing programming, data parallel programming, message passing interface (MPI), and parallel virtual machine. Also, topics relating to modes of communication between processors, the functions defined in MPI and PVM are discussed in sufficient details.

2.5 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1) ***hello_other.c***

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the “slave” or spawned program; its first PVM action is to obtain the task id of the ``master'' using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and

architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

2.6 REFERENCES/FURTHER READINGS

1) Kai Hwang: *Advanced Computer Architecture*: Parallelism, Scalability, Programmability (2001), Tata McGraw Hill, 2001.

2) Henessy J. L. and Patterson D. A. *Computer Architecture*: A Qualititative Approach, Morgan Kaufman (1990)

3) Rajaraman V. and Shive Ram Murthy C. *Parallel Computer: Architecture and Programming*: Prentice Hall of India

4) Salim G. *Parallel Computation, Models and Methods*: Akl Prentice Hall of India

UNIT 3 PARALLEL PROGRAMMING

3.0 INTRODUCTION

After getting a great breakthrough in the serial programming and figuring out its limitations, computer professionals and academicians are focusing now on parallel programming. Parallel programming is a popular choice today for multi-processor architectures to solve the complex problems. If developments in the last decade give any indications, then the future belongs to of parallel computing. Parallel programming is intended to take advantages the of non-local resources to save time and cost and overcome the memory constraints.

In this section, we shall introduce parallel programming and its classifications. We shall discuss some of the high level programs used for parallel programming. There are certain compiler-directive based packages, which can be used along with some high level languages. We shall also have a detailed look upon them.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the basics of parallel programming;
- describe the parallel programming based on message passing;
- learn programming using High Performance Fortran, and
- learn compiler directives of OpenMP.

3.2 INTRODUCTION TO PARALLEL PROGRAMMING

Traditionally, a software has been written for serial computation in which programs are written for computers having a single Central Processing Unit (CPU). Here, the problems are solved by a series of instructions, executed one after the other, one at a time, by the CPU. However, many complex, interrelated events happening at the same time like planetary and galactic orbital events, weather and ocean patterns and tectonic plate drift may require super high complexity serial software. To solve these large problems and save the computational time, a new programming paradigm called parallel programming was introduced.

To develop a parallel program, we must first determine whether the problem has some part which can be parallelised. There are some problems like generating the Fibonacci Sequence in the case of which there is a little scope for parallelization. Once it has been determined that the problem has some segment that can be parallelized, we break the problem into discrete chunks of work that can be distributed to multiple tasks. This partition of the problem may be data-centric or function-centric. In the former case, different functions work with different subsets of data while in the latter each function performs a portion of the overall work. Depending upon the type of partition approach, we require communication among the processes. Accordingly, we have to design the mode of communication and mechanisms for process synchronization.

3.3 TYPES OF PARALLEL PROGRAMMING

There are several parallel programming models in common use. Some of these are:

- Message Passing;
- Data Parallel programming;
- Shared Memory; and
- Hybrid.

In the last unit, we had a brief introduction about these programming models. In this unit we shall discuss the programming aspects of these models.

3.3.1 Programming Based on Message Passing

As we know, the programming model based on message passing uses high level programming languages like C/C++ along with some message passing libraries like MPI and PVM. We had discussed MPI and PVM at great length in unit 2 on PRAM algorithms. Hence we are restricting ourselves to an example program of message passing.

Example 1: Addition of array elements using two processors.

In this problem, we have to find the sum of all the elements of an array A of size n. We shall divide n elements into two groups of roughly equal size. The first [n/2] elements are added by the first processor, P_0, and last [n/2] elements the by second processor, P_1. The two sums then are added to get the final result. The program is given below:

Program for P_0

```
#include <mpi.h>
#define n 100
int main(int argc, char **argv) {
int A[n];
int sum0 =0, sum1=0,sum;
MPI_Init(&argc, &argv);
for( int i=0;i<n;i++)
scanf("%d", &A[i]);
MPI_Send( &n/2, n/2, MPI_INT,1, 0, MPI_COMM_WORLD);
for(i=1; i<n/2; i++)
 sum0+=A[i];
sum1=MPI_Recv(&n/2, n/2, MPI_INT,1, 0, MPI_COMM_WORLD);
sum=sum0+sum1;
printf("%d",sum);
MPI_Finalize();
}
```

Program for P_1,

```
int func( int B[int n])
{
MPI_Recv(&n/2, n/2, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i</2; i++)
  sum1+=B[i];
MPI_Send( 0, n/2, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

☞ **Check Your Progress 1**

1) Enumerate at least five applications of parallel programming.
……
……
……
……

2) What are the different steps to write a general parallel program?
……
……
……
……

3) Write a program to find the sum of the elements of an array using k processors.
……
……
……
……

3.3.2 Programming Based on Data Parallelism

In a data parallel programming model, the focus is on data distribution. Each processor works with a portion of data. In the last unit we introduced some data parallel languages. Now, we shall discuss one of the most popular languages for parallel programming based on data parallel model.

High Performance FORTRAN

In 1993 the *High Performance FORTRAN Forum*, a group of many leading hardware and software vendors and academicians in the field of parallel processing, established an informal language standard called *High Performance Fortran* (HPF). It was based on Fortran 90, then it extended the set of parallel features, and provided extensive support for computation on *distributed memory* parallel computers. The standard was supported by a majority of vendors of parallel hardware.

HPF is a highly suitable language for data parallel programming models on MIMD and SIMD architecture. It allows programmers to add a number of compiler directives that minimize inter-process communication overhead and utilize the load-balancing techniques.

We shall not discuss the complete HPF here, rather we shall focus only on augmenting features like:

- Processor Arrangements,
- Data Distribution,
- Data Alignment,
- FORALL Statement,
- INDEPENDENT loops, and
- Intrinsic Functions.

3.3.2.1 Processor Arrangements

It is a very frequent event in data parallel programming to group a number of processors to perform specific tasks. To achieve this goal, HPF provides a directive called *PROCESSORS* directive. This directive declares a conceptual processor grid. In other words, the PROCESSORS directive is used to specify the shape and size of an array of abstract processors. These do not have to be of the same shape as the underlying hardware.The syntax of a PROCESSORS directive is:

!HPF$ PROCESSORS array_name (dim1, dim 2, ….dim n)

where array_name is collective name of abstract processors. dim i specifies the size of i^{th} dimension of array_name.

Example 2:

!HPF$ PROCESSORS P (10)
This introduces a set of 10 abstract processors, assigning them the collective name `P`.

!HPF$ PROCESSORS Q (4, 4)
It introduces 16 abstract processors in a 4 by 4 array.

3.3.2.2 Data Distribution

Data distribution directives tell the compiler how the program data is to be distributed amongst the memory areas associated with a set of processors. The logic used for data distribution is that if a set of data has independent sub-blocks, then computation on them can be carried out in parallel. They do not allow the programmer to state directly which processor will perform a particular computation. But it is expected that if the operands of

a particular sub-computation are all found on the same processor, the compiler will allocate that part of the computation to the processor holding the operands, whereupon no remote memory accesses will be involved.

Having seen how to define one or more target processor arrangements, we need to introduce mechanisms for distributing the data arrays over those arrangements. The DISTRIBUTE directive is used to distribute a data object) onto an abstract processor array.

The syntax of a DISTRIBUTE directive is:

!HPF$ DISTRIBUTE array_lists [ONTO arrayp]

where array_list is the list of array to be distributed and arrayp is abstract processor array.

The ONTO specifier can be used to perform a distribution across a particular processor array. If no processor array is specified, one is chosen by the compiler.

HPF allows arrays to be distributed over the processors directly, but it is often more convenient to go through the intermediary of an explicit *template*. A template can be declared in much the same way as a processor arrangement.

```
!HPF$ TEMPLATE T(50, 50, 50)
```

declares a 50 by 50 by 50 three-dimensional template called T. Having declared it, we can establish a relation between a template and some processor arrangement by using DISTRIBUTE directive. There are three ways in which a template may be distributed over Processors: *Block, cyclic and **.

(a) Block Distribution

Simple block distribution is specified by

!HPF$ DISTRIBUTE T1(BLOCK) ONTO P1
where T1 is some template and P1 is some processor arrangement.
In this case, each processor gets a contiguous block of template elements. All processors get the same sized block. The last processor may get lesser sized block.

Example 3:

```
!HPF$ PROCESSORS P1(4)
!HPF$ TEMPLATE T1(18)
!HPF$ DISTRIBUTE T1(BLOCK) ONTO P1
```

As a result of these instructions, distribution of data will be as shown in *Figure 1*.

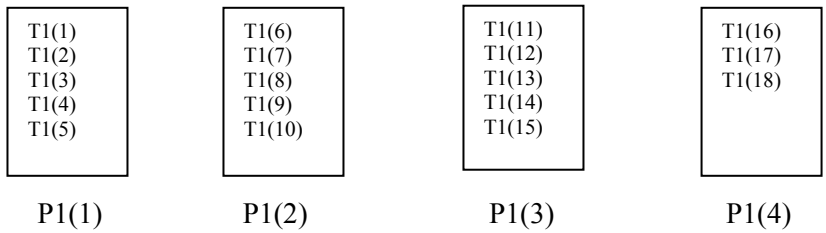


Figure 1: Block Distribution of Data

In a variant of the block distribution, the number of template elements allocated to each processor can be explicitly specified, as in
!HPF$ DISTRIBUTE T1 (BLOCK (6)) ONTO P1

Distribution of data will be as shown in *Figure 2*.

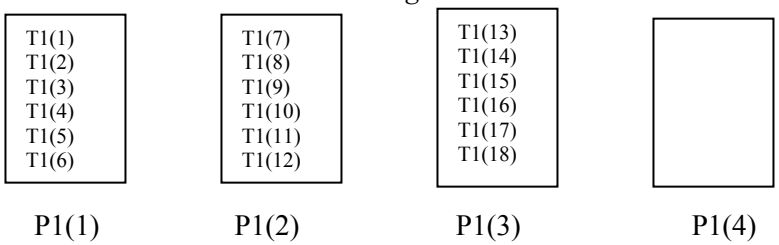


Figure 2: Variation of Block Distribution

It means that we allocate all template elements before exhausting processors, some processors are left empty.

(b) Cyclic Distribution

Simple cyclic distribution is specified by

!HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1

The first processor gets the first template element, the second gets the second, and so on. When the set of processors is exhausted, go back to the first processor, and continue allocating the template elements from there.

Example 4

!HPF$ PROCESSORS P1(4)
!HPF$ TEMPLATE T1(18)
!HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1

The result of these instructions is shown in *Figure 3*.

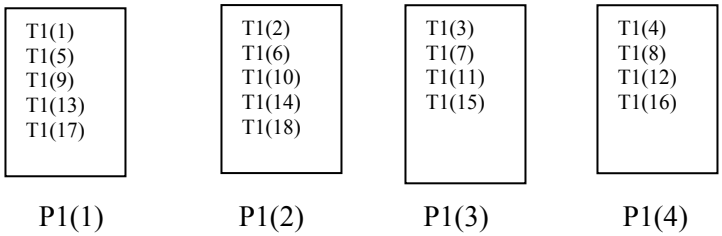


Figure 3: Cyclic Distribution

But in an analogous variant of the cyclic distribution
!HPF$ DISTRIBUTE T1 (BLOCK (3)) ONTO P1

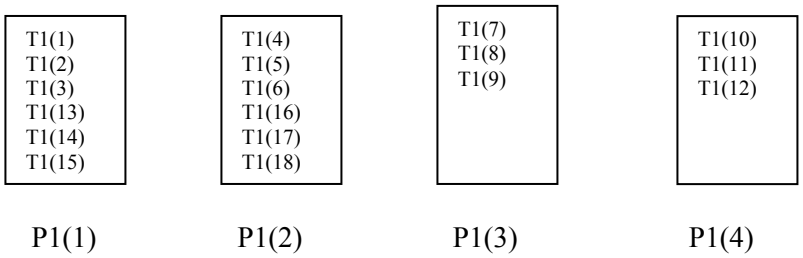


Figure 4: Variation of Cyclic Distribution

That covers the case where both the template and the processor are one dimensional. When the and processor have (the same) higher dimension, each dimension can be distributed independently, mixing any of the four distribution formats. The correspondence between the template and the processor dimension is the obvious one. In
!HPF$ PROCESSORS P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (CYCLIC, BLOCK) ONTO P2
the first dimension of T2 is distributed cyclically over the first dimension of P2; the second dimension is distributed blockwise over the second dimension of P2.

*(c) * Distribution*

Some dimensions of a template may have ``collapsed distributions'', allowing a template to be distributed onto a processor arrangement with fewer dimensions than the template.

Example 5

```
!HPF$ PROCESSORS P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (BLOCK, *) ONTO P1
```

means that the first dimension of `T2` will be distributed over `P1` in blockwise order but for a fixed value of the first index of `T2`, all values of the second subscript are mapped to the same processor.

3.3.2.3 Data Alignment

Arrays are aligned to templates through the ALIGN directive. The `ALIGN` directive is used to align elements of different arrays with each other, indicating that they should be distributed in the same manner. The syntax of an ALIGN derivative is:

!HPF$ ALIGN array1 WITH array2

where array1 is the name of array to be aligned and array2 is the array to be aligned to.

Example 6

Consider the statement
ALIGN A[i] WITH B[i]

This statement aligns each A[i] with B[i] as shown in *Figure 5*.

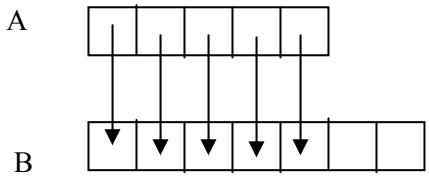


Figure 5: ALIGN A[i] WITH B[i]

Consider the statement

ALLIGN A[i] WITH B[i+1]

This statement aligns the each A[i] with B[i+1] as shown in *Figure 6*.

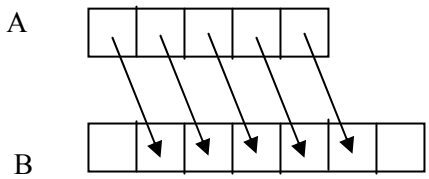


Figure 6: ALIGN A[i] WITH B[i+1]

* can be used to collapse dimensions. Consider the statement

ALLIGN A[:, *] WITH B[:]

This statement aligns two dimensional array with one dimensional array by collapsing as shown in *Figure 7*.

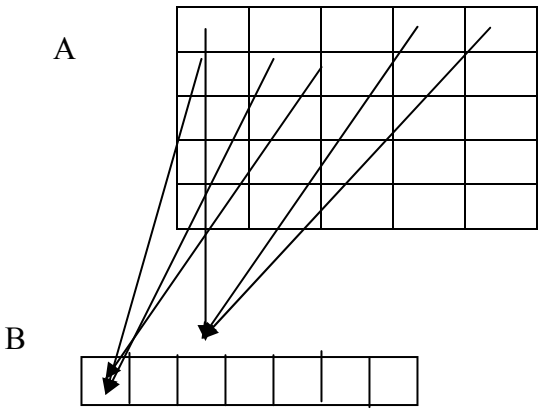


Figure 7: * alignment

3.3.2.4 The FORALL Statement

The FORALL statement allows for more general assignments to sections of an array. A FORALL statement has the general form.

FORALL (*triplet*, ..., *triplet*, *mask*)
statement

where *triplet* has the general form

subscript = *lower*: *upper* : *step-size*

and specifies a set of indices. step-size is optional. *statement* is an arithmetic or pointer assignment and the assignment statement is evaluated for those index values specified by the list of triplets that are not rejected by the optional *mask*.

Example 7 The following statements set each element of matrix `X` to the sum of its indices.

FORALL (i=1:m, j=1:n) X(i,j) = i+j

and the following statement sets the upper right triangle of matrix `Y to` zero .

FORALL (i=1:n, j=1:n, i<j) Y(i,j) = 0.0

Multi-statement FORALL construct:
Multi-statement FORALL is shorthand for the series of single statement FORALLs. The syntax for FORALL is

FORALL (index-spec-list [,mask])
Body
END FORALL

Nesting of FORALL is allowed.

Example 8
Let a= [2,4,6,8,10], b=[1,3,5,7,9], c=[0,0,0,0,0]
Consider the following program segment

```
FORALL (i = 2:4)
  a(i) = a(i-1)+a(i+1)
  c(i) = b(i) *a(i+1).
END FORALL
```

The computation will be

```
  a[2] =a[1]+a[3] =2+6=8
  a[3] =a[2]+a[4] =4+8=12
  a[4] =a[3]+a[5] = 6+10=16
  c[2] = b[2] *a[3] = 3*12=36
  c[3] = b[3] *a[4] = 5*16=80
  c[4] = b[4] *a[5] =7*10=70
```

Thus output is
a= [2,8,12,16,10], b=[1,3,5,7,9], c=[0,36,80,70,0]

3.3.2.5 INDEPENDENT Loops

HPF provides additional opportunities for parallel execution by using the INDEPENDENT directive to assert that the iterations of a do-loop can be performed independently---that is, in any order or concurrently---without affecting the result . In effect, this directive changes a do-loop from an implicitly parallel construct to an explicitly parallel construct.

The INDEPENDENT directive must immediately precede the do-loop to which it applies. In its simplest form, it has no additional argument and asserts simply that no iteration of the do-loop can affect any other iteration.

Example 9

In the following code fragment, the directives indicate that the outer two loops are independent. The inner loop assigns the elements of `A` repeatedly and hence it is not independent.

```
!HPF$   INDEPENDENT

do i=1,n1  ! Loop over  i independent

  !HPF$   INDEPENDENT

  do j=1,n2      ! Loop over  j independent

    do k=1,n3          ! Inner loop not independent

      A(i,j) = A(i,j) + B(i,j,k)*C(i,j)

          enddo

  enddo

enddo
```

3.3.2.6 Intrinsic Functions

HPF introduces some new intrinsic functions in addition to those defined in F90. The two most frequently used in parallel programming are the system inquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE. These functions provide the information about the *number* of physical processors on which the running program executes and processor configuration. General syntax of is

NUMBER_OF_PROCESSORS is

NUMBER_OF_PROCESSORS (dim)

where `dim` is an optional argument. It returns the number of processors in the underlying array or, if the optional argument is present, the size of this array along a specified dimension.

General syntax of PROCESSORS_SHAPE is

PROCESSORS_SHAPE()

It returns an one-dimensional array, whose i^{th} element gives the size of the underlying processor array in its i^{th} dimension.

Example 10

Consider the call of the two intrinsic functions discussed above for a 32-Processor (4×8) Multicomputer:

The function call NUMBER_OF_PROCESORS () will return 32.
The function call NUMBER_OF_PROCESORS (1) will return 4.
The function call NUMBER_OF_PROCESORS (2) will return 8.
The function call PROCESSORS_SHAPE () will return an array with two elements 4 and 8.

We can use these intrinsic functions in tandem with array declarations and HPF directives, to provide flexibility to the programmer to declare abstract processor arrays that match available physical resources. For example, the following statement !HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()) declares an abstract processor array `P` with size equal to the number of physical processors.

☞ Check Your Progress 2

1) Give the output of the following instructions in HPF:
   (a) !HPF$ PROCESSORS Q (s, r)
   (b) !HPF$ PROCESSORS P1(5)
   !HPF$ TEMPLATE T1(22)
   !HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1.

2) Write a FORALL statement to set lower triangle of a matrix X to zero.
   ……
   ……
   ……
   ……

3) What are intrinsic functions? Name any two of them.
   ……
   ……
   ……
   ……

3.3.3 Shared Memory Programming

As discussed in unit 2, we know that all processors share a common memory in shared memory model. Each processor, however, can be assigned a different part of the program stored in the memory to execute with the data stored in specified locations. Each processor does computation independently with the data allocated to them by the controlling program, called the main program. After finishing their computations, all of these processors join the main program. The main program finishes only after all child processes have been terminated completely. There are many alternatives to implement these ideas through high level programming. Some of them are:

i) Using heavy weight processes.
ii) Using threads.(e.g. Pthreads).
iii) Using a completely new programming language for parallel programming (e.g. Ada).
iv) Using library routines with an existing sequential programming language.
v) Modifying the syntax of an existing sequential programming language to create a parallel programming language (e.g. UPC).

(vi) Using an existing sequential programming language supplemented with the compiler directives for specifying parallelism (e.g. OpenMP).

We shall adopt the last alternative. We shall also give an introduction to the shared programming using library routines with an existing sequential programming language.

3.3.3.1 OpenMP

OpenMP is a compiler directive based standard developed in the late 1990s jointly by a group of major computer hardware and software vendors. It is portable across many popular platforms including Unix and Windows NT platforms. The OpenMP Fortran API was released on October 28, 1997 and the C/C++ API was released in late 1998. We shall discuss only about C/C++ API.

The OpenMP API uses the fork-join model of parallel execution. As soon as an OpenMP program starts executing it creates a single thread of execution, called the initial thread. The initial thread executes sequentially. As soon as it gets a *parallel* construct, the thread creates additional threads and works as the master thread for all threads. All of the new threads execute the code inside the *parallel* construct. Only the master thread continues execution of the user code beyond the end of the *parallel* construct. There is no restriction on the number of *parallel* constructs in a single program. When a thread with its child threads encounters a work-sharing construct, the work inside the construct is divided among the members of the team and executed co-operatively instead of being executed by every thread. Execution of the code by every thread in the team resumes after the end of the work-sharing construct. Synchronization constructs and the library routines are available in OpenMP to co-ordinate threads and data in *parallel* and work-sharing constructs.

Each OpenMP directive starts with *#pragma omp*. The general syntax is

#pragma omp directive-name [Set of clauses]

where *omp* is an OpenMP keyword. There may be additional clauses (parameters) after the directive name for different options.

Now, we shall discuss about some compiler directives in OpenMP.

(i) Parallel Construct

The syntax of the *parallel* construct is as follows:

#pragma omp parallel [set of clauses]
where *clause* is one of the following:
structured-block
if(scalar-expression)
private(list)
firstprivate(list)
default(shared | none)
shared(list)
copyin(list)

When a thread encounters a *parallel* construct, a set of new threads is created to execute the *parallel* region. Within a parallel region each thread has a unique thread number. Thread number of the master thread is zero. Thread number of a thread can be obtained by

the call of library function *omp_get_thread_num***.** Now, we are giving the description of the clauses used in a parallel construct.

(a) Private Clause:

This clause declares one or more list items to be private to a thread. The syntax of the *private* clause is

private(list).

(b) Firstprivate Clause:

The *firstprivate* clause declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered. The syntax of the *firstprivate* clause is as follows:

*firstprivate***(***list***).**

(c) Shared Clause:

The *shared* clause declares one or more list items to be shared among all the threads in a team. The syntax of the *shared* clause is :

*shared***(***list***)**

(d) Copyin Clause:

The *copyin* clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region. The syntax of the *copyin* clause is :

*copyin***(***list***)**

(ii) Work-Sharing Constructs

A work-sharing construct distributes the execution of the associated region among the members of the team that encounters it. A work-sharing construct does not launch new threads.

OpenMP defines three work-sharing constructs: *sections, for, and single*.
In all of these constructs, there is an implicit barrier at the end of the construct unless a *nowait* clause is included.

(a) Sections

The *sections* construct is a no iterative work-sharing construct that causes the structured blocks to be shared among the threads in team. Each structured block is executed once by one of the threads in the team. The syntax of the *sections* construct is:

```
#pragma omp sections [set of clauses.]
{
 #pragma omp section
  structured-bloc
#pragma omp section
structured-block
.
.
.
}
```

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

(i) Lastprivate Clause

The *lastprivate* clause declares one or more list items to be private to a thread, and causes the corresponding original list item to be updated after the end of the region. The syntax of the *lastprivate* clause is:
Lastprivate (*list*)

(ii) Reduction Clause

The *reduction* clause specifies an operator and one or more list items. For each list item, a private copy is created on each thread, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator. The syntax of the *reduction* clause is :

reduction (*operator***:***list*)

(b) For Loop Construct

The loop construct causes the for loop to be divided into parts and parts shared among threads in the team. The syntax of the loop construct is :
#*pragma omp for [set of clauses.]*
for-loop

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)

(c) Single Construct

The *single* construct specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The other threads in the team do not execute the block, and wait at an implicit barrier at the end of the *single* construct, unless a *nowait* clause is specified.The syntax of the *single* construct is as follows:

#pragma omp single [set of clauses]
structured-block

The *clause* is one of the following:
private(list)
firstprivate(list)
copyprivate(list)
nowait

(iii) Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are shortcuts for specifying a work-sharing construct nested immediately inside a *parallel* construct. The combined parallel work-sharing constructs allow certain clauses which are permitted on both *parallel* constructs and on work-sharing constructs. OpenMP specifies the two combined parallel work-sharing constructs: *parallel loop* construct, and *parallel sections* construct.

(a) Parallel Loop Construct

The parallel loop construct is a shortcut for specifying a *parallel* construct containing one loop construct and no other statements. The syntax of the parallel loop construct is :
#pragma omp parallel for [set of clauses]
for-loop

(a) Parallel Sections Construct

The *parallel sections* construct is a shortcut for specifying a *parallel* construct containing one *sections* construct and no other statements. The syntax of the *parallel sections* construct is:
#pragma omp parallel sections [set of clauses]
{
[#pragma omp section]
structured-block
[#pragma omp section
structured-block]
...
}
In the following example, routines *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first *section* directive is optional. Note that all the *section* directives need to appear in the *parallel sections* construct.

(iv) Master Construct

The master directive has the following general form:

#pragma omp master
structured_block

It causes the master thread to execute the structured block. Other threads encountering this directive will ignore it and the associated structured block, and will move on. In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

(v) Critical Directive

The *critical* directive allows one thread execute the associated structured block. When one or more threads reach the critical directive, they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block. The syntax of the critical directive is

#pragma omp critical [name]
structured_block

name is optional. All critical sections with no name are considered to be one undefined name.

(vi) Barrier Directive

The syntax of the barrier directive is
#pragma omp barrier
When a thread reaches the barrier it waits until all threads have reached the barrier and then they all proceed together. There are restrictions on the placement of barrier directive in a program. The *barrier* directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax.

(vii) Atomic Directive

The *atomic* directive ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of atomic directive is:

#pragma omp atomic
expression_statement

The atomic directive implements a critical section efficiently when the critical section simply updates a variable by arithmetic operation defined by expression_statement.

(viii) Ordered directive

This directive is used in conjunction with *for* and *parallel for* directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop. The syntax of the *ordered* construct is as follows:
#pragma omp ordered new-line
structured-block

3.3.3.2 Shared Programming Using Library Routines

The most popular of them is the use of combo function called fork() and join(). Fork() function is used to create a new child process. By calling join() function parent process waits the terminations of the child process to get the desired result.

Example 11: Consider the following set of statements

```
Process A                    Process B

  :                            :

fork B ;                       :

 :                             :

join B;                      end B;
```

In the above set of statements process A creates a child process B by the statement fork B. Then A and B continue their computations independently until A reaches the join statement, At this stage, if B is already finished, then A continues executing the next statement otherwise it waits for B to finish.

In the shared memory model, a common problem is to synchronize the processes. It may be possible that more than one process are trying to simultaneously modify the same variable. To solve this problem many synchronization mechanism like test_and_set, semaphores and monitors have been used. We shall not go into the details of these mechanisms. Rather, we shall represent them by a pair of two processes called lock and unlock. Whenever a process P locks a common variable, then only P can use that variable. Other concurrent processes have to wait for the common variable until P calls the unlock on that variable. Let us see the effect of locking on the output of a program when we do not use lock and when we use lock.

Example 12

Let us write a pseudocode to find sum of the two functions f(A) + f(B). In the first algorithm we shall not use locking.

```
Process A                      Process B

 sum = 0                           :

    :                              :

  fork B                        sum = sum+ f(B)

   :                               :

 sum = sum + f(A)                  end B

    :

   join B

   :

 end A
```

If process A executes the statement sum = sum + f(A) and writes the results into main memory followed by the computation of sum by process B, then we get the correct result. But consider the case when B executes the statement sum = sum + f(B) before process A could write result into the main memory. Then the sum contains only f(B) which is incorrect. To avoid such inconsistencies, we use locking.

```
Process A                      Process B

 sum = 0                           :

    :                              :

    :                              lock sum

  fork B                        sum = sum + f(B)

   :                               unlock sum
```

```
lock sum                        :

 sum = sum + f(A)               end B

unlock sum

  :

   join B

  :

 end A
```

In this case whenever a process acquires the sum variable, it locks it so that no other process can access that variable which ensures the consistency in results.

3.3.4 Example Programmes for Parallel Systems

Now we shall finish this unit with the examples on shared memory programming.

Example 13: Adding elements of an array using two processor

```
 int sum, A[ n] ;  //shared variables
void main ( ){

                int i ;

        for (i=0; i<n; i++)
        scanf ("%d",&A[i] );
        sum=0;
        // now create process to be executed by processor P1
        fork(1) add (A,n/2,n-1, sum); // process to add elements from index n/2 to -
        1.sum is output variable        // now create process to be executed by processor
        P0                                                      add (A,0,n/2-1,
        sum);
        join 1 ;
        printf ("%d", sum);

        }

 add (int A[ ], int lower, int upper, int sum) {

                        int sum1=0, i;
        for (i=lower; i<=upper; i++)
        sum1=sum1+A[i];
        lock sum;
        sum=sum+sum1;
        unlock  sum ;
        }
```

In this program, the last half of the array is passed to processor P_1 which adds them. Meanwhile processor P_0 adds the fist half of the array. The variable sum is locked to avoid inconsistency.

Example 14: In this example we will see the use of parallel construct with private and firstprivate clauses. At the end of the program i and j remain undefined as these are private to thread in parallel construct.

```
#include <stdio.h>
int main()
{
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
i = 3;
j = j + 2;
}
printf("%d %d\n", i, j); /* i and j are undefined */
return 0;
}
```

In the following example, each thread in the **parallel** region decides what part of the global array *x* to work on, based on the thread number:

Example 15

```
#include <omp.h>
void subdomain(float x[ ], int istart, int ipoints)
{
int i;
for (i = 0; i < ipoints; i++)
x[istart+i] = 123.456;
}
void sub(float x[ 10000], int npoints)
{
int t_num, num_t, ipoints, istart;
#pragma omp parallel default(shared) private(t_num , num_t, ipoints, istart)
{
t_num = omp_get_thread_num(); //thread number of current thread
num_t = omp_get_num_threads(); //number of threads
ipoints = npoints / num_t; /* size of partition */
istart = t_num * ipoints; /* starting array index */
if (t_num == num_t-1) /* last thread may do more */
ipoints = npoints - istart;
subdomain(x, istart, ipoints);
}
}

int main()
{
float array[10000];
sub(array, 10000);
```

```
return 0;
}
```

In this example we used two library methods : *omp_get_num_threads()* and *omp_get_thread_num()*.
omp_get_num_threads() returns number of threads that are currently being used in parallel directive.
omp_get_thread_num() returns thread number (an integer from 0 to omp_get_num_threads() - 1 where thread 0 is the master thread).

Example 16

This example illustrate the use of lastprivate clause

```
void for_loop (int n, float *a, float *b)
{
int i;
#pragma omp parallel
{
#pragma omp for lastprivate(i)
for (i=0; i<n-1; i++)
a[i] = b[i] + b[i+1];
}
a[i]=b[i];   /* i == n-1 here */
}
```

Example 17

This example demonstrates the use of parallel sections construct. The three functions, fun1, fun2, and fun3, all can be executed concurrently. Note that all the section directives need to appear in the parallel sections construct.

```
void fun1();
void fun2();
void fun3();
void  parallel_sec()
{
#pragma omp parallel sections
{
#pragma omp section
fun1();
#pragma omp section
fun2();
#pragma omp section
fun3();
}
}
```

☞ **Check Your Progress 3**

1) Write the syntax of the following compiler directives in OpenMP:
   (a) Parallel
   (b) Sections
   (c) Master

2) What do you understand by synchronization of processes? Explain at least one mechanism for process synchronisation.
   ……
   ……
   ……
   ……

3) Write a shared memory program to process marks of the students. Your program should take the roll number as input and the marks of students in 4 different subjects and find the grade of the student, class average and standard deviation.
   ……
   ……
   ……
   ……

3.4 SUMMARY

In this unit, the following four types of models of parallel computation are discussed in detail:

- Message Passing;
- Data Parallel programming;
- Shared Memory; and
- Hybrid.

Programming based on message passing has already been discussed in Unit 2. In context of data parallel programming, various issues related to High Performance Fortran, e.g., data distribution, block distribution, cyclic distribution, data alignment etc are discussed in sufficient details. Next, in respect of the third model of parallel computation, viz Shared Memory, various constructs and features provided by the standard OpenMP are discussed. Next, in respect of this model, some issues relating to programming using library routines are discussed.

3.5 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1) Application of parallel programming:

   i) In geology and metrology, to solve problems like planetary and galactic orbits, weather and ocean patterns and tectonic plate drift;
   ii) In manufacturing to solve problems like automobile assembly line;
   iii) In science and technology for problems like chemical and nuclear reactions, biological, and human genome;
   iv) Daily operations within a business; and
   v) Advanced graphics and virtual reality, particularly in the entertainment industry.

2) Steps to write a parallel program:

i) Understand the problem thoroughly and analyze that portion of the program that can be parallelized;
ii) Partition the problem either in data centric way or in function centric way depending upon the nature of the problem;
iii) Decision of communication model among processes:
iv) Decision of mechanism for synchronization of processes,
v) Removal of data dependencies (if any),
vi) Load balancing among processors,
vii) Performance analysis of program.

3) Program for P_0

```
#include <mpi.h>
#define n 100
int main(int argc, char **argv) {
int A[n];
int sum0 =0, sum1[ ],sum, incr, last_incr;
MPI_Init(&argc, &argv);
for( int i=0;i<n;i++)   //taking input array
scanf("%d", &A[i]);
incr = n/k;     //finding number of data for each processor
last_incr = incr n + n%k;  // last processor may get lesser number of data
for(i=1; i<= k-2; i++)
MPI_Send ( (i-1)*incr, incr,  MPI_INT,i, i, MPI_COMM_WORLD); // P0 sends
data to other processors
MPI_Send( (i-1)*incr, last_incr,  MPI_INT,i, i, MPI_COMM_WORLD); // P0
sends data to last processor
for(i=1; i<=incr; i++)    // P0 sums its own elements
 sum0+=A[i];
for(i=1; i<= k-1; i++)          // P0 receives results from other processors
sum1[i] =MPI_Recv(i, 1, MPI_INT,0, 0, MPI_COMM_WORLD);
for(i=1; i<= k-1; i++)                    //results are added to get final results
sum=sum0+sum1[i];
printf("%d",sum);
MPI_Finalize();
}
```

// Program for P_r for r=1 2 …k-2,

```
int func( int B[int n])
{
MPI_Recv(1, incr, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i<incr; i++)
  sum1+=B[i];
MPI_Send( 0, incr, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

Program for the last processor Pk_{-1},

```
int func( int B[int n])
{
```

```
MPI_Recv(1, last_incr, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i<last_incr; i++)
  sum1+=B[i];
MPI_Send( 0, last_incr, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

☞ Check Your Progress 2

1) a) !HPF$ PROCESSORS Q (s, r)
It maps s × r processors along a two dimensional array and gives them collective name Q.

b) !HPF$ PROCESSORS P(5)
!HPF$ TEMPLATE T(22)
!HPF$ DISTRIBUTE T(CYCLIC) ONTO P.
Data is distributed n 5 processors as shown below:

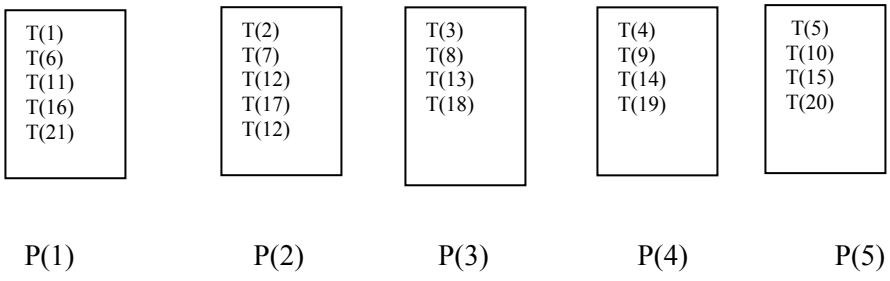


Figure 8

2) FORALL (i=1:n, j=1:n, i >j) Y(i,j) = 0.0

3) Intrinsic functions are library-defined functions in a programming languages to support various constructs in the language. The two most frequently used in parallel programming are the system inquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE. These functions provide information about the number of physical processors on which the running program executes and processor configuration. General syntax of NUMBER_OF_PROCESSORS is

NUMBER_OF_PROCESSORS(dim)

where dim is an optional argument. It returns the number of processors in the underlying array or, if the optional argument is present, the size of this array along a specified dimension.

General syntax of PROCESSORS_SHAPE is

PROCESSORS_SHAPE()

It returns an one-dimensional array, whose i^{th} element gives the size of the underlying processor array in its i^{th} dimension.

☞ Check Your Progress 3

1) (a) syntax for parallel directive :
#pragma omp parallel [set of clauses]
where *clause* is one of the following:
structured-block
if(scalar-expression)
private(list)
firstprivate(list)
default(shared | none)
shared(list)
copyin(list)

(b) syntax for sections directive :

#pragma omp sections [set of clauses.]
{
#pragma omp section
structured-bloc
#pragma omp section
structured-block
.
.
.
}

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

(c) syntax for master directive :

#pragma omp master
structured_block

2) In parallel programming, shared memory programming in particular, processes very often compete with each other for common resources. To ensure the consistency, we require some mechanisms called process synchronization. There are numerous techniques to enforce synchronization among processes. We discussed one of them in unit 3.2.2 using lock and unlock process. Now we are introducing semaphores. Semaphores were devised by Dijkstra in 1968. It consists of two operations P and V operating on a positive integer s (including zero).P waits until *s* is greater than zero and then decrements *s* by one and allows the process to continue. V increments *s* by one and releases one of the waiting processes (if any). **P** and **V** operations are performed atomically. Mechanism for activating waiting processes is also implicit in **P** and **V** operations. Processes delayed by **P**(s) are kept in abeyance until released by a **V**(s) on the same semaphore. Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable.

3)
```
#include <math.h>

int total_m, num_stud=0,sum_marks, sum_square;

void main ( )

{
int roll_no, marks[ ][4],i ;
char grade;
float class_av, std_dev;
while (!EOF) {
scanf (“%d”, &roll_no);
for (i=0; i<4; i++)          //taking marks of individual student
scanf (“%d”,&marks[num_stud][i] );
total_m =0;
num_stud++;
for (i=0; i<4; i++)
total_m = total_m+marks[num_stud][i];   //sum of marks
fork(1) grade = find_grade();    //create new parallel process to find grade
fork(2) find_stat();                  // create  new parallel process to  find sum and
squares
join 1;             //wait until process find_grade terminates
join 2;             // wait until process find_stat terminates
printf (“roll number %d”, roll_no);
for (i=0; i<4; i++)
printf (“%d”,marks[num_stud][i] );
printf (“%d”,total_m );
printf (“%c”,grade );
}
class_av= sum_marks/num_stud;
std_dev=sqrt ((sum_square/std_num) –(class_av*class_av)) ;
printf (“%f %f”,class_av,std_dev );
}

char find_grade( ) {
char g;
if  (total_m >80)    g=’E’;
else if   (total_m >70)    g=’A’;
else if   (total_m >60)    g=’B’;
else if   (total_m >50)    g=’C’;
else if   (total_m >40)    g=’D’;
else   g=’F’;
return g;
}

find_stat( ) {
sum_marks =sum_marks+total_m;
sum_square =sum_square+total_m*total_m;
}
```

Example 18: master construct

```
#include <stdio.h>
extern float average(float,float,float);
void master_construct ( float* x, float* xold, int n, float tol )
```

```
{
int c, i, toobig;
float error, y;
c = 0;
#pragma omp parallel
{
do{
#pragma omp for private(i)
for( i = 1; i < n-1; ++i ){
xold[i] = x[i];
}
#pragma omp single
{
toobig = 0;
}
#pragma omp for private(i,y,error) reduction(+:toobig)
for( i = 1; i < n-1; ++i ){
y = x[i];
x[i] = average( xold[i-1], x[i], xold[i+1] );
error = y - x[i];
if( error > tol || error < -tol ) ++toobig;
}
#pragma omp master
{
++c;
printf( "iteration %d, toobig=%d\n", c, toobig );
}
}while( toobig > 0 );
}
}
```

3.6 REFERENCES/FURTHER READINGS

1) Quinn Michael J. *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Education (2004).

2) Jorg Keller, Kesler Christoph W. and Jesper Larsson *Practical PRAM Programming* (wiley series in Parallel Computation).

3) Ragsdale & Susan, (ed) "*Parallel Programming*" McGraw-Hill.

4) Chady, K. Mani & Taylor slephen "*An Introduction to Parallel Programming*, Jones and Bartleh.

UNIT 1 OPERATING SYSTEM FOR PARALLEL COMPUTER

Structure **Page Nos.**

1.0 INTRODUCTION

In Blocks 1 and 2, we have discussed parallel computing architectures and parallel algorithms. This unit discusses the additional requirements at operating system and software levels which will make the parallel programs run on parallel hardware. Collectively, these requirements define the parallel program development environment. A parallel programming environment consists of available hardware, supporting languages, operating system along with software tools and application programs. The hardware platforms have already been discussed in the earlier units. These topics include discussion of shared memory systems, message passing systems, vector processing; scalar, superscalar, array and pipeline processors and dataflow computers. This unit also presents a case study regarding operating systems for parallel computers.

1.1 OBJECTIVES

After studying this unit you will be able to describe the features of software and operating systems for parallel computers.

In particular you should be able to explain the following:

- various additional requirements imposed at OS level for parallel computer systems;
- parallel programming environment characteristics;
- multitasking environment, and
- features of Parallel UNIX.

1.2 PARALLEL PROGRAMMING ENVIRONMENT CHARACTERISTICS

The parallel programming environment consists of an editor, a debugger, performance evaluator and programme visualizer for enhancing the output of parallel computation. All programming environments have these tools in one form or the other. Based on the features of the available tool sets the programming environments are classified as basic, limited, and well developed. The Basic environment provides simple facilities for program tracing and debugging. The limited integration facilities provide some additional tools for parallel debugging and performance evaluation. Well-developed environments provide most advanced tools of debugging programs, for textual graphics interaction and for parallel graphics handling.

There are certain parallel overheads associated with parallel computing. The parallel overhead is the amount of time required to coordinate parallel tasks, as opposed to doing useful work. These include the following factors:

i) Task start up time
ii) Synchronisations
iii) Data communications.

Besides these hardware overheads, there are certain software overheads imposed by parallel compilers, libraries, tools and operating systems.

The parallel programming languages are developed for parallel computer environments. These are developed either by introducing new languages (e.g. occam) or by modifying existing languages like, (FORTRAN and C). Normally the language extension approach is preferred by most computer designs. This reduces compatibility problem. High-level parallel constructs were added to FORTRAN and C to make these languages suitable for parallel computers. Besides these, optimizing compilers are designed to automatically detect the parallelism in program code and convert the code to parallel code.

In addition to development of languages and compilers for parallel programming, a parallel programming environment should also have supporting tools for development and text editing of parallel programmes.

Let us now discuss the examples of parallel programming environments of Cray Y-MP software and Intel paragaon XP/S.

The Cray Y-MP system works with UNICOS operating system. It has two FORTRAN compilers CFT 77 and CFT for automatic vector code generation. The system software has large library of routines, program management utilities, debugging aids and assembler UNICOS written in C. It supports optimizing, vectorizing, concurrentising facilities for FORTRAN compilers and also has optimizing and vetorizing C compiler. The Cray Y-MP has three multiprocessing/multitasking methods namely, (i) macrotasking, (ii) microtasking, (iii) autotasking. Also, it has a subroutine library, containing various utilities, high performance subroutines along with math and scientific routines.

The Intel Paragaon XP/S system is an extension of Intel iPSC/860 and Delta systems, and is a scalable and mesh connected multicompiler which is implemented in a distributed memory system.

The processors that for nodes of the system are 50 MHz i860 XP Processors. Further, it uses distributed UNIX based OS technology. The languages supported by Paragaon include C, C++, Data Parallel Fortran and ADA. The tools for integration include FORGE and Cast parallelisation tools. The programming environment includes an Interactive Parallel Debugger (IPD).

1.3 SYNCHRONIZATION PRINCIPLES

In multiprocessing, various processors need to communicate with each other. Thus, synchronisation is required between them. The performance and correctness of parallel execution depends upon efficient synchronisation among concurrent computations in multiple processes. The synchronisation problem may arise because of sharing of writable data objects among processes. Synchronisation includes implementing the order of operations in an algorithm by finding the dependencies in writable data. Shared object access in an MIMD architecture requires dynamic management at run time, which is much more complex as compared to that of SIMD architecture. Low-level synchronization primitives are implemented directly in hardware. Other resources like CPU, Bus and memory unit also need synchronisation in Parallel computers.

To study the synchronization, the following dependencies are identified:

i) **Data Dependency:** These are WAR, RAW, and WAW dependency.

ii) **Control dependency:** These depend upon control statements like GO TO, IF THEN, etc.

iii) **Side Effect Dependencies:** These arise due to exceptions, Traps, I/O accesses.

For the proper execution order as enforced by correct synchronization, program dependencies must be analysed properly. Protocols like wait protocol, and sole access protocol are used for doing synchronisation.

1.3.1 Wait protocol

The wait protocol is used for resolving the conflicts, which arise because of a number of multiprocessors demanding the same resource. There are two types of wait protocols: busy-wait and sleep-wait. In busy-wait protocol, process stays in the process context register, which continuously tries for processor availability. In sleep-wait protocol, wait protocol process is removed from the processor and is kept in the wait queue. The hardware complexity of this protocol is more than busy-wait in multiprocessor system; if locks are used for synchronization then busy-wait is used more than sleep-wait.

Execution modes of a multiprocessor: various modes of multiprocessing include parallel execution of programs at (i) Fine Grain Level (Process Level), (ii) Medium Grain Level (Task Level), (iii) Coarse Grain Level (Program Level).

For executing the programs in these modes, the following actions/conditions are required at OS level.

i) Context switching between multiple processes should be fast. In order to make context switching easy multiple sets should be present.

ii) The memory allocation to various processes should be fast and context free.

iii) The Synchronization mechanism among multiple processes should be effective.

iv) OS should provide software tools for performance monitoring.

1.3.2 Sole Access Protocol

The atomic operations, which have conflicts, are handled using sole access protocol. The method used for synchronization in this protocol is described below:

1) **Lock Synchronization:** In this method contents of an atom are updated by requester process and sole access is granted before the atomic operation. This method can be applied for shared read-only access.

2) **Optimistic Synchronization:** This method also updates the atom by requester process, but sole access is granted after atomic operation via abortion. This technique is also called post synchronisation. In this method, any process may secure sole access after first completing an atomic operation on a local version of the atom, and then executing the global version of the atom. The second operation ensures the concurrent update of the first atom with the updation of second atom.

3) **Server synchronization:** It updates the atom by the server process of requesting process. In this method, an atom behaves as a unique update server. A process requesting an atomic operation on atom sends the request to the atom's update server.

Check Your Progress 1

1) In sleep-wait synchronization mechanism, we know a process is removed/suspended from the processor and put in a wait queue. Suggest some fairness policies for reviving removed/suspended process

……
……
……
……

1.4 MULTI TASKING ENVIRONMENT

Multi tasking exploits parallelism by:

1) Pipelining functional units are pipe line together
2) Concurrently using the multiple functional units
3) Overlapping CPU and I/O activities.

In multitasking environment, there should be a proper mix between task and data structures of a job, in order to ensure their proper parallel execution.

In multitasking, the useful code of a programme can be reused. The property of allowing one copy of a programme module to be used by more than task in parallel is called reentrancy. Non-reentrant code can be used only once during lifetime of the programme. The reentrant codes, which may be called many times by different tasks, are assigned with local variables.

Shared variable programme structures

The concept of shared variable has already been discussed above. In this section, we discuss some more concepts related to the shared programme.

1.4.1 Concept of Lock

Locks are used for protected access of data in a shared variable system. There are various types of locks:

1) **Binary Locks:** These locks are used globally among multiple processes.

2) **Deckkers Locks:** These locks are based on distributed requests, to ensure mutual exclusion without unnecessary waiting.

1.4.2 System Deadlock

A deadlock refers to the situation when concurrent processes are holding resources and preventing each other from completing their execution.

The following conditions can avoid the deadlock from occurring:

1) **Mutual exclusion:** Each process is given exclusive control of the resources allotted to it.

2) **Non-preemption:** A process is not allowed to release its resources till task is completed.

3) **Wait for:** A process can hold resources while waiting for additional resources.

4) **Circular wait:** Multiple processes wait for resources from the other processes in a circularly dependent situation.

1.4.3 Deadlock Avoidance

To avoid deadlocks two types of strategies are used:

1) **Static prevention:** It uses P and V operators and Semaphores to allocate and deallocate shared resources in a multiprocessor. Semaphores are developed based on sleep wait protocol. The section of programme, where a deadlock may occur is called critical section. Semaphores are control signals used for avoiding collision between processes.

   P and V technique of Deadlock prevention associates a Boolean value 0 or 1 to each semaphore. Two atomic operators, P and V are used to access the critical section represented by the semaphore. The P(s) operation causes value of semaphore s to be increased by one if s is already at non-zero value.

   The V(s) operation increases the value of s by one if it is not already one. The equation s=1 indicates the availability of the resource and s=0 indicates non-availability of the resource.

   During execution, various processes can submit their requests for resources asynchronously. The resources are allocated to various processors in such a way that

they do not create circular wait. The shortcoming of the static prevention is poor resource utilisation.

2) Another method of deadlock prevention is **dynamic deadlock avoidance**. It checks deadlocks on runtime condition, which may introduce heavy overhead in detecting potential existence of deadlocks.

Check Your Progress 2

1) Explain the spin lock mechanism for synchronisation among concurrent processes. Then define a binary spin lock.

……
……
……
……

1.5 MESSAGE PASSING PROGRAMME DEVELOPMENT ENVIRONMENT

In a multicomputer system, the computational load between various processors must be balanced. To pass information between various nodes, message-passing technique is used. The programming environment of a multicomputer includes a host runtime system and resident operating system called Kernel in all the node computers. The host system provides uniform communication between processes without intervention of the nodes, host workstation and network connection. Host processes are located outsides the nodes. The host environment for hypercube computer at Caltech is a UNIX processor and uses UNIX and language processor utilities to communicate with node processes. The Kernel is separately located in each node computer that supports multiprogramming with an address space confined by local memory. Many node processes can be created at each node. All node processes execute concurrently in different physical node or interleaved through multiprogramming within the same node. Node processes communicate with each other by sending or receiving messages.

The messages can be of various types. A particular field of all messages can be reserved to represent message type. The message passing primitives are as follows:

- Send (type, buffer, length, node process)
- Receive (type, buffer, length)

Where type identifies the message type, buffer indicates location of the message, length specifies the length of the message, node designates the destination node and process specifies process ID at destination node. Send and receive primitives are used for denoting the sending and receiving processes respectively. The buffer field of send specifies the memory location from which messages are sent and the buffer field of receive indicates the space where arriving messages will be stored.
The following issues are decided by the system in the process of message passing:

1) Whether the receiver is ready to receive the message
2) Whether the communication link is established or not
3) One or more messages can be sent to the same destination node

The message passing can be of two types: Synchronous and Asynchronous

In **Synchronous**, the message passing is implemented on synchronous communication network. In this case, the sender and receiver processes must be synchronized in time and space. Time synchronization means both processes must be ready before message transmission takes place. The space synchronization demands the availability of interconnection link.

In **Asynchronous** message passing, message-sending ands receiving are not synchronized in time and space. Here, the store and forward technology may be used. The blocked messages are buffered for later transmission. Because of finite size buffer the system may be blocked even in buffered Asynchronous non-blocking system.

1.6 UNIX FOR MULTIPROCESSOR SYSTEM

The UNIX operating system for a multiprocessor system has some additional features as compared to the normal UNIX operating system. Let us first discuss the design goals of the multiprocessor UNIX. The original UNIX developed by Brian Kernighan and Dennis Ritchie was developed as portable, general purpose, time-sharing uniprocessor operating system.

The OS functions including processor scheduling, virtual memory management, I/O devices etc, are implemented with a large amount of system software. Normally the size of the OS is greater than the size of the main memory. The portion of OS that resides in the main memory is called kernel. For a multiprocessor, OS is developed on three models viz: Master slave model, floating executive model, multithreaded kernel. These UNIX kernels are implemented with locks semaphores and monitors.
Let us discuss these models in brief.

1) **Master slave kernel:** In this model, only one of the processors is designated as Master.

   The master is responsible for the following activities:

   i) running the kernel code

   ii) handling the system calls

   iii) handling the interrupts.

   The rest of the processors in the system run only the user code and are called slaves.

2) **Floating-Executive model:** The master-slave kernel model is too restrictive in the sense that only one of the processors viz the designated master can run the kernel. This restriction may be relaxed by having more than one processors capable of running the kernel and allowing additional capability by which the master may float among the various processors capable of running the kernel.

3) **Multi-threaded UNIX kernel:** We know that threads are light-weight processors requiring minimal state information comprising the processor state and contents of relevant registers. A thread being a (light weight) process is capable of executing alone. In a multiprocessor system, more than one processors may execute simultaneously with each processor possibly executing more than one threads, with

the restriction that those threads which share resources must be allotted to one processor. However, the threads which do not share resources may be allotted to different processors. In this model, in order to separate multiple threads requiring different sets of kernel resources spin locks or semaphores are used.

1.7 SUMMARY

In this unit, various issues relating to operating systems and other software requirements for parallel computers are discussed.

In parallel systems, the issue of synchronisation becomes very important, specially in view of the fact more than one processes may require the same resource at same point of time. For the purpose, synchronisation principles are discussed in section 1.3. For the purpose of concurrent sharing of the memory in multitasking environment, the concepts of lock, deadlock etc are discussed in section 1.4. Finally, extensions of UNIX for multiprocessor systems are discussed in section 1.6.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Out of large number of fairness policies, the following three policies are quite well known:

   1) First-In-First-Out (FIFO) policy
   2) Upper-bounded misses policy
   3) Livelock-free policy

   First-In-First-Out (FIFO) as the name suggests that no process will be served out of turn. The disadvantage is that in some cases cost of computation and memory requirement may be too high.

   The second fairness policy may be implemented in a number of ways. Depending upon the implementation, the implementation costs and memory requirements may be reduced.

Check Your Progress 2

1) Under the centralized shared (CS) memory, the gate for entry and exist to CS is controlled by a single binary variable say y, which is then shared by all processes attempting to access CS.

   For defining the binary spin lock
   Let y be the single variable for entry to the gate. Initially the value of y is set to 0 (zero) indicating that entry by any one process is allowed. Then spin lock controlling mechanism continuously checks one by one all processes in turn, whether any one of these processes needs to access the memory. Once, any of the processes is found need to access CS, that process say Pp is allowed to access the CS and the entry-allowing variable y is assigned 0 (zero) indicating next process in queue requiring access to CS is allowed to access CS.

1.9 FURTHER READINGS

1) Kai Hwang: *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, Tata-McGraw-Hill (2001)

2) Thomas L. Casavant, Pavel Tvrdik, Frantisck Plasil, *Parallel Computers: Theory and Practice*, IEEE

UNIT 2 PERFORMANCE EVALUATIONS

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier blocks and the previous section of this block, we have discussed a number of types of available parallel computer systems. They have differences in architecture, organisation, interconnection pattern, memory organisation and I/O organisation. Accordingly, they exhibit different performances and are suitable for different applications. Hence, certain parameters are required which can measure the performance of these systems.

In this unit, the topic of performance evaluation explains those parameters that are devised to measure the performances of various parallel systems. Achieving the highest possible performance has always been one of the main goals of parallel computing. Unfortunately, most often the real performance is less by a factor of 10 and even worse as compared to the designed peak performance. This makes parallel performance evaluation an area of priority in high-performance parallel computing. As we already know, sequential algorithms are mainly analyzed on the basis of computing time i.e., time complexity and this is directly related to the data input size of the problem. For example, for the problem of sorting n numbers using bubble sort, the time complexity is of $O(n^2)$. However, the performance analysis of any parallel algorithm is dependent upon three major factors viz. time complexity, total number of processors required and total cost. The complexity is normally related with input data size (n).

Thus, unlike performance of a sequential algorithm, the evaluation of a parallel algorithm cannot be carried out without considering the other important parameters like the total number of processors being employed in a specific parallel computational model. Therefore, the evaluation of performance in parallel computing is based on the parallel computer system and is also dependent upon machine configuration like PRAM,

combinational circuit, interconnection network configuration etc. in addition to the parallel algorithms used for various numerical as well non-numerical problems.

This unit provides a platform for understanding the performance evaluation methodology as well as giving an overview of some of the well-known performance analysis techniques.

2.1 OBJECTIVES

After studying this unit, you should be able to:

- describe the Metrics for Performance Evaluation;
- tell about various Parallel System Overheads;
- explain the speedup Law; and
- enumerate and discuss Performance Measurement Tools.

2.2 METRICS FOR PERFORMANCE EVALUATION

In this section, we would highlight various kinds of metrics involved for analysing the performance of parallel algorithms for parallel computers.

2.2.1 Running Time

The running time is the amount of time consumed in execution of an algorithm for a given input on the N-processor based parallel computer. The running time is denoted by T(n) where n signifies the number of processors employed. If the value of n is equal to 1, then the case is similar to a sequential computer. The relation between Execution time vs. Number of processors is shown in *Figure 1*.

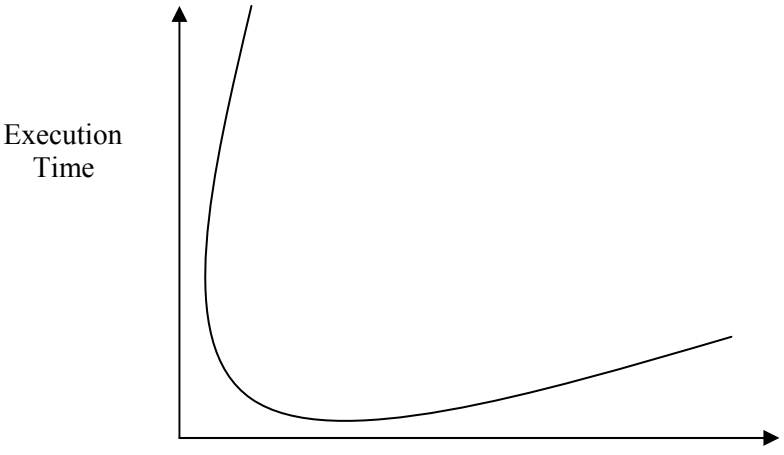


Figure 1: Execution Time vs. number of processors

It can be easily seen from the graph that as the number of processors increases, initially the execution time reduces but after a certain optimum level the execution time increases as number of processors increases. This discrepancy is because of the overheads involved in increasing the number of processes.

2.2.2 Speed Up

Speed up is the ratio of the time required to execute a given program using a specific algorithm on a machine with single processor (i.e. T (1) (where n=1)) to the time required

to execute the same program using a specific algorithm on a machine with multiple processors (i.e. T(n)). Basically the speed up factor helps us in knowing the relative gain achieved in shifting from a sequential machine to a parallel computer. It may be noted that the term T(1) signifies the amount of time taken to execute a program using the best sequential algorithm i.e., the algorithm with least time complexity.

$$S(n) = \frac{T(1)}{T(n)}$$

Let us take an example and illustrate the practical use of speedup. Suppose, we have a problem of multiplying n numbers. The time complexity of the sequential algorithm for a machine with single processor is O(n) as we need one loop for reading as well as computing the output. However, in the parallel computer, let each number be allocated to individual processor and computation model being used being a hypercube. In such a situation, the total number of steps required to compute the result is log n i.e. the time complexity is O(log n). *Figure 2,* illustrates the steps to be followed for achieving the desired output in a parallel hypercube computer model.

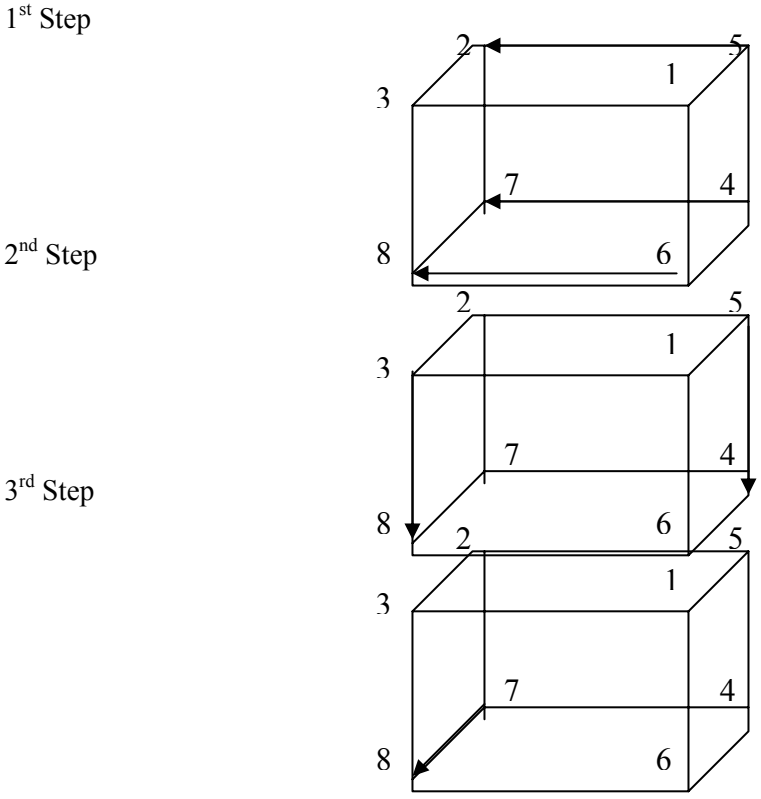


Figure 2: Steps followed for multiplying n numbers stored on n processors

As the number of steps followed is equal to 3 i.e., log 8 where n is number of processors. Hence, the complexity is O(log n).

In view of the fact that sequential algorithm takes 8 steps and the above-mentioned parallel algorithm takes 3 steps, the speed up is as under:

$$S(n) = \frac{8}{3}$$

As the value of S(n) is inversely proportional to the time required to compute the output on a parallel number which in turn is dependent on number of processors employed for performing the computation. The relation between S(n) vs. Number of processors is shown in *Figure 3*. The speedup is directly proportional to number of processors,

therefore a linear arc is depicted. However, in situations where there is parallel overhead, the arc is sub-linear.

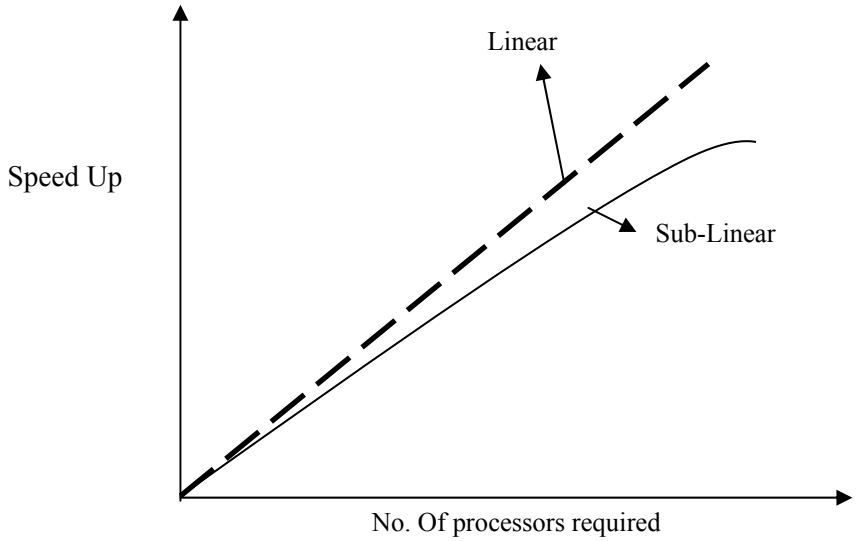


Figure 3: Speed-up vs. Number of Processors

2.2.3 Efficiency

The other important metric used for performance measurement is efficiency of parallel computer system i.e. how the resources of the parallel systems are being utilized. This is called as degree of effectiveness. The efficiency of a program on a parallel computer with *k* processors can be defined as the ratio of the relative speed up achieved while shifting the load from single processor machine to *k* processor machine where the multiple processors are being used for achieving the result in a parallel computer. This is denoted by E(k).

E_k is defined as follows:

$$E(k) = \frac{S(k)}{k}$$

The value of E(k) is directly proportional to S(k) and inversely proportional to number of processors employed for performing the computation. The relation between E(k) vs. Number of processors is shown in *Figure 4*.

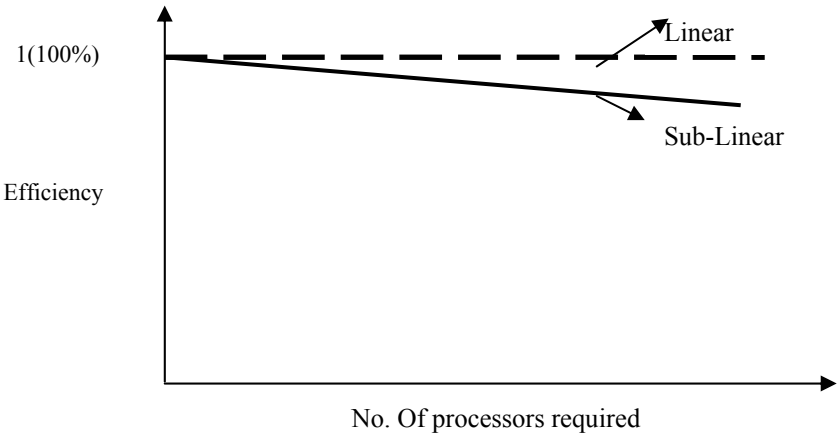


Figure 4: Efficiency vs. Number of Processors

Assuming we have the multiplication problem as discussed above with *k* processors, then the efficiency is as under:

$$E(k) = \frac{T(1)}{T(n) * K} = \frac{N}{\log n * N}$$

$$E(n) = \frac{1}{\log(n)}$$

Now, supposing we have X processors i.e. X < K and we have to multiply n numbers, in such a situation the processors might be overloaded or might have a few overheads. Then the efficiency is as under:

$$E(X) = \frac{T(1)}{T(X) * X}$$

Now, the value of T(X) has to be computed. As we have n numbers, and we have X processors, therefore firstly each processor will multiply *n/X* numbers and consequently process the X partial results on the X processors according to the method discussed in *Figure 1*. The time complexity is equal to the sum of the time to compute multiplication of k/X numbers on each processor i.e., O(k/X) and time to compute the solution of partial results i.e. log (X)

$$E(X) = \frac{K}{(K / X + \log(X)) * X}$$

$$E(X) = \frac{(K / X)}{(K / X + \log(X))}$$

Dividing by X/K we get

$$E(X) = \frac{1}{(1 + (X / K) * \log(X))}$$

It can be concluded from the above statement that if N is fixed then the efficiency i.e. E(X) will decrease as the value of X increases and becomes equivalent to E(N) in case X=N. Similarly, if X is fixed then the efficiency i.e., E(X) will increase as the value of X i.e. the number of computations increases.

The other performance metrics involve the standard metrics like MIPS and Mflops. The term MIPS (Million of Instructions Per Second) indicates the instruction execution rate. Mflops (Million of Floating Point Operations per Second) indicates the floating-point execution rate.

2.3 FACTOR CAUSING PARALLEL OVERHEADS

Figures 2,3 and *4* clearly illustrate that the performance metrics are not able to achieve a linear curve in comparison to the increase in number of processors in the parallel computer. The reason for the above is the presence of overheads in the parallel computer which may degrade the performance. The well-known sources of overheads in a parallel computer are:

1) Uneven load distribution
2) Cost involved in inter-processor communication
3) Synchronization
4) Parallel Balance Point

The following section describes them in detail.

2.3.1 Uneven Load Distribution

In the parallel computer, the problem is split into sub-problems and is assigned for computation to various processors. But sometimes the sub-problems are not distributed in a fair manner to various sets of processors which causes imbalance of load between various processors. This event causes degradation of overall performance of parallel computers.

2.3.2 Cost Involved in Inter-Processor Communication

As the data is assigned to multiple processors in a parallel computer while executing a parallel algorithm, the processors might be required to interact with other processes thus requiring inter-processor communication. Therefore, there is a cost involved in transferring data between processors which incurs an overhead.

2.3.3 Parallel Balance Point

In order to execute a parallel algorithm on a parallel computer, K number of processors are required. It may be noted that the given input is assigned to the various processors of the parallel computer. As we already know, execution time decreases with increase in number of processors. However, when input size is fixed and we keep on increasing the number of processors, in such a situation after some point the execution time starts increasing. This is because of overheads encounteed in the parallel system.

2.3.4 Synchronization

Multiple processors require synchronization with each other while executing a parallel algorithm. That is, the task running on processor X might have to wait for the result of a task executing on processor Y. Therefore, a delay is involved in completing the whole task distributed among K number of processors.

Check Your Progress 1

1) Which of the following are the reasons for parallel overheads?
   A) Uneven load distribution
   B) Cost involved in inter-processor communication
   C) Parallel Balance Point
   D) All of the above

2) Which of the following are the performance metrics?
   A) MIPS
   B) Mflops
   C) Parallel Balance Point
   D) A and B only

3) Define the following terms:
   A) Running Time
   B) Speed Up
   C) Efficiency

2.4 LAWS FOR MEASURING SPEED UP PERFORMANCE

To measure speed up performance various laws have been developed. These laws are discussed here.

2.4.1 Amdahl's Law

Remember, the speed up factor helps us in knowing the relative gain achieved in shifting the execution of a task from sequential computer to parallel computer and the performance does not increase linearly with the increase in number of processors. Due to the above reason of saturation in 1967 Amdahl's law was derived. Amdahl's law states that a program contains two types of operations i.e. complete sequential operations which must be done sequentially and complete parallel operations which can be executed on multiple processors. The statement of Amdahl's law can be explained with the help of an example.

Let us consider a problem say P which has to be solved using a parallel computer. According to Amdahl's law, there are mainly two types of operations. Therefore, the problem will have some sequential operations and some parallel operations. We already know that it requires T (1) amount of time to execute a problem using a sequential machine and sequential algorithm. The time to compute the sequential operation is a fraction α (alpha) ($\alpha \leq 1$) of the total execution time i.e. T (1) and the time to compute the parallel operations is (1- α). Therefore, S (N) can be calculated as under:

$$S(N) = \frac{T(1)}{T(N)}$$

$$S(N) = \frac{T(1)}{X * T(1) + (1-\alpha) * \frac{T(1)}{N}}$$

Dividing by T(1)

$$S(N) = \frac{1}{\alpha + \frac{1-\alpha}{N}}$$

Remember, the value of α is between 0 and 1. Now, let us put some values of α and compute the speed up factor for increasing values of number of processors. We find that the S(N) keeps on decreasing with increase in the value of α (i.e. number of sequential operations as shown in *Figure 5*).

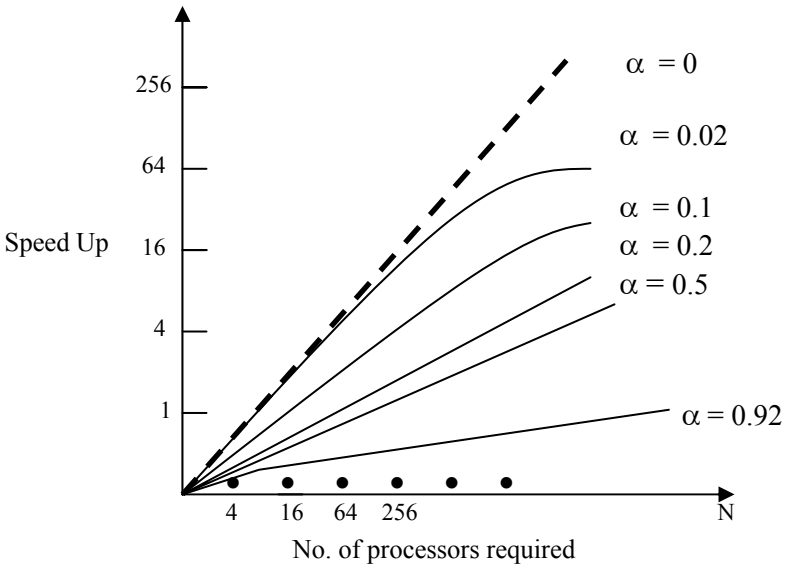


Figure 5: Speed-up vs. Number of Processors

The graph in *Figure 5* clearly illustrates that there is a bottleneck caused due to sequential operations in a parallel computer. Even when the number of sequential operations are more, after increasing the number of processors, the speed up factor S (N) degrades.

The sequential fraction i.e. α can be compared with speed up factor S(N) for a fixed value of N say 500. *Figure 6* illustrates a pictorial view of effect of Amdahl's law on the speed up factor.

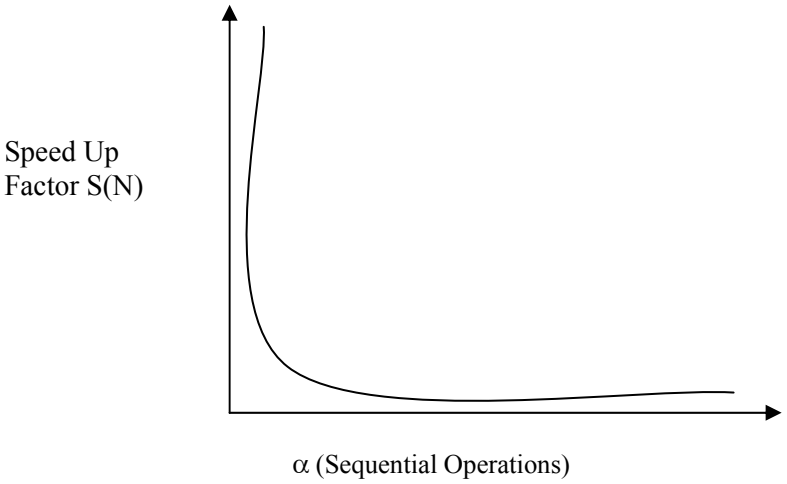


Figure 6: S (n) vs. α (Graph is not to scale)

The outcomes of analysis of Amdahl's law are:

1) To optimize the performance of parallel computers, modified compilers need to be developed which should aim to reduce the number of sequential operations pertaining to the fraction α.
2) The manufacturers of parallel computers were discouraged from manufacturing large-scale machines having millions of processors.

There is one major shortcoming identified in Amdahl's law. According to Amdahl's law, the workload or the problem size is always fixed and the number of sequential operations mainly remains same. That is, it assumes that the distribution of number of sequential operations and parallel operations always remains same. This situation is shown in

Figure 7 and the ratio of sequential and parallel operations are independent of problem size.

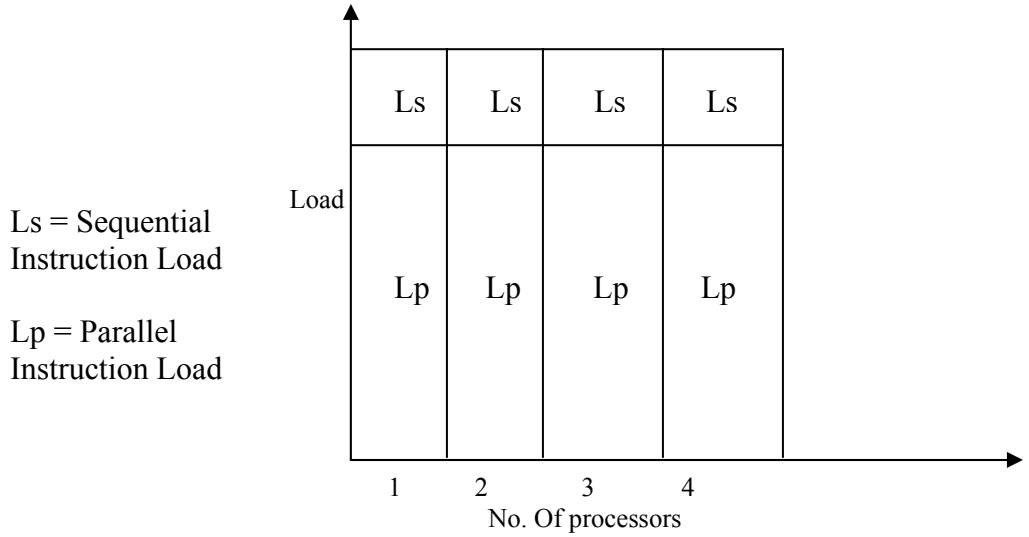


Figure 7: Fixed load for Amdahl's Law

However, practically the number of parallel operations increases according to the size of problem. As the load is assumed to be fixed according to Amdahl's law, the execution time will keep on decreasing when number of processors is increased. This situation is shown in *Figure 8*. This is in Introduction to the processes operation.

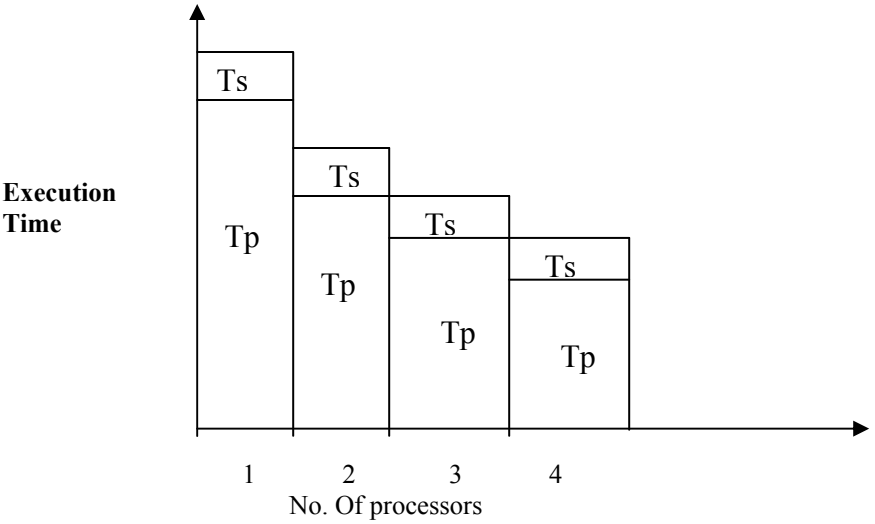


Figure 8: Execution Time decreases for Amdahl's Law

2.4.2 Gustafson's Law

Amdahl's law is suitable for applications where response time is critical. On the other hand, there are many applications which require that accuracy of the resultant output should be high. In the present situation, the computing power has increased substantially due to increase in number of processors attached to a parallel computer. Thus it is possible to increase the size of the problem i.e., the workload can be increased. How does this operate? Gustafson's Law relaxed the restriction of fixed size of problem and aimed at using the notion of constant execution time in order to overcome the sequential bottleneck encountered in Amdahl's Law. This situation which does not assume a fixed workload is analysed by Gustafson. Thus, Gustafson's Law assumes that the workload may increase substantially, with the number of processors but the total execution time should remain the same as highlighted in *Figures 9* and *10.*

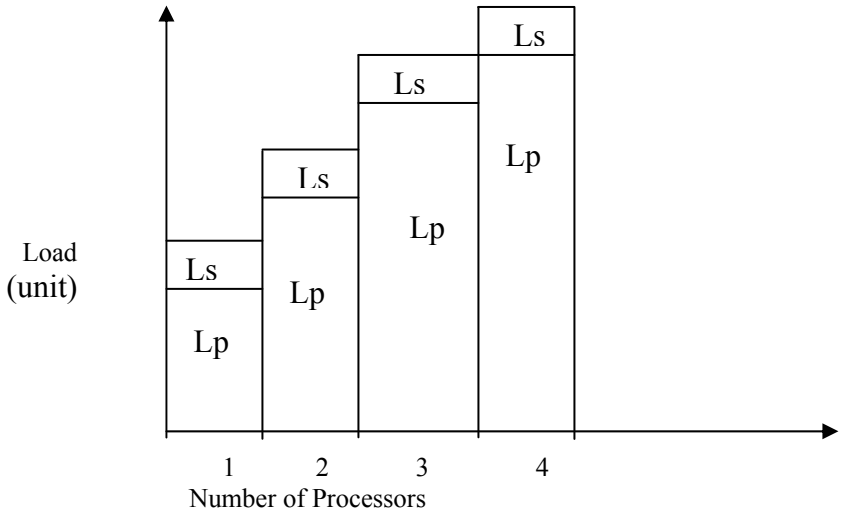


Figure 9: Parallel Load Increases for Gustafson's Law

According to Gustafson's Law, if the number of parallel operations for a problem increases sufficiently, then the sequential operations will no longer be a bottleneck.

The statement of Gustafson's law can be explained with the help of an example. Let us consider a problem, say P, which has to be solved using a parallel computer. Let T_s be the time taken which is considered as constant for executing sequential operations. Let T_p (N, L) be the time taken for running the parallel operations with L as total load on a parallel computer with N processors. The total time taken for finding the solution of problem is T=Ts + Tp. Therefore, S (N) can be calculated as under:

$$S(N) = \frac{T(1)}{T(N)}$$

$$S(N) = \frac{T_s + T_p(1, L)}{T_s + T_p(N, L)}$$

Also, Tp (1, L)= N Tp (N, L) i.e. time taken to execute parallel operations having a load of L on one processor is equal to the N multiplied by the time taken by one computer having N processor. If α be the fraction of sequential load for a given problem i.e.,*

$$\alpha = \frac{T_s}{T_s + T_p(N, L)}$$

Substituting Tp (1, L)= N Tp (N, L) we get,*

$$S(n) = \frac{T_s + N * T_p(N, L)}{T_s + T_p(N, L)}$$

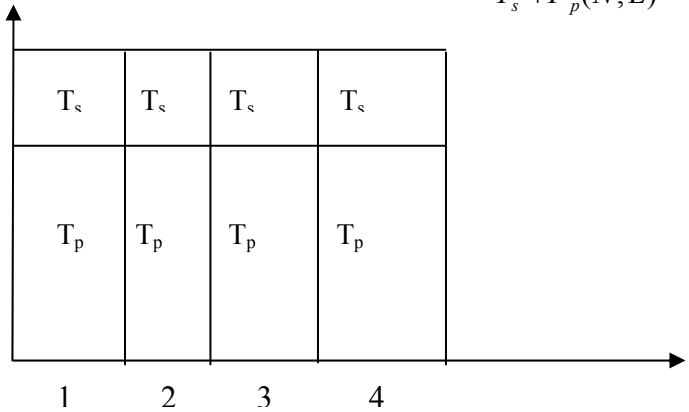


Figure 10: Fixed Execution Time for Gustafson's Law

$$S(N) = \frac{T_s}{T_s + T_p(N,L)} + \frac{N * T_p(N,L)}{T_s + T_p(N,L)}$$

Now, let us change the complete equation of S(N) in the form of X. We get

*S(N) = α +N * (1-α)*

S(N) = N –α * (N-1)

Now, let us put some values of α and compute the speed up factor for increasing values of α i.e. sequential factor of work load for a fixed number of processors say N. *Figure 11* illustrates a pictorial view of effect of Gustafson's Law on the speed up factor. The graph shows as the value of α increases, the speed up factor increases. This decrease is because of overhead caused by inter-processor communication.

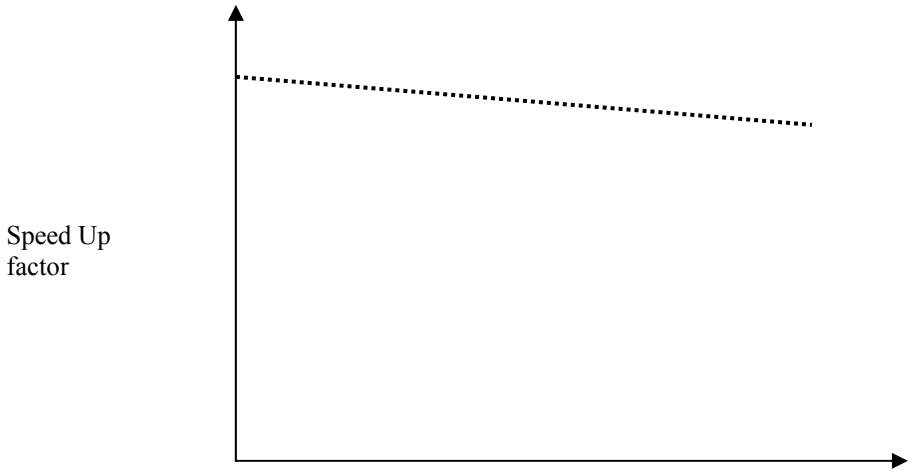


Figure 11: S (N) vs. α (Graph is not to scale)

2.4.3 Sun and Ni's Law

The Sun and Ni's Law is a generalisation of Amdahl's Law as well as Gustafson's Law. The fundamental concept underlying the Sun and Ni's Law is to find the solution to a problem with a maximum size along with limited requirement of memory. Now-a-days, there are many applications which are bounded by the memory in contrast to the processing speed.

In a multiprocessor based parallel computer, each processor has an independent small memory. In order to solve a problem, normally the problem is divided into sub-problems and distributed to various processors. It may be noted that the size of the sub-problem should be in proportion with the size of the independent local memory available with the processor. Now, for a given problem, when large set of processors is culminated together, in that case the overall memory capacity of the system increases proportionately. Therefore, instead of following Gustafson's Law i.e., fixing the execution time, the size of the problem can be increased further such that the memory could be utilized. The above technique assists in generating more accurate solution as the problem size has been increased.

State the law and then inter part discuss it.

Check Your Progress 2

1) Which of the following laws deals with finding the solution of a problem with maximum size along with limited requirement of memory?

   a) Amdahl's law
   b) Gustafson's law
   c) Sun and Ni's law
   d) None of the above

2) Which of the following laws state the program, which contains two types of operations i.e. complete sequential operations which must be done sequentially and complete parallel operations which can be executed on multiple processors?

   a) Amdahl's law
   b) Gustafson's law
   c) Sun and Ni's law
   d) None of the above

3) Which of the following law used the notion of constant execution time?

   a) Amdahl's law
   b) Gustafson's law
   c) Sun and Ni's law
   d) None of the above

2.5 TOOLS FOR PERFORMANCE MEASUREMENT

In the previous units, we have discussed various parallel algorithms. The motivation behind these algorithms has been to improve the performance and gain a speed up. After the parallel algorithm has been written and executed, the performance of both the algorithms is one of the other major concerns. In order to analyze the performance of algorithms, there are various kinds of performance measurement tools. The measurement tools rely not only on the parallel algorithm but require to collect data from the operating system and the hardware being used, so as to provide effective utilisation of the tools.

For a given parallel computer, as the number of processors and its computational power increases, the complexity and volume of data for performance analysis substantially increases. The gathered data for measurement is always very difficult for the tools to store and process it.

The task of measuring the performance of the parallel programs has been divided into two components.

1) **Performance Analysis**: It provides the vital information to the programmers from the large chunk of statistics available of the program while in execution mode or from the output data.

2) **Performance Instrumentation:** Its emphasis on how to efficiently gather information about the computation of the parallel computer.

2.6 PERFORMANCE ANALYSIS

In order to measure the performance of the program, the normal form of analysis of the program is to simply calculate the total amount of CPU time required to execute the various part of the program i.e., procedures. However, in case of a parallel algorithm running on a parallel computer, the performance metric is dependent on several factors such as inter-process communication, memory hierarchy etc. There are many tools such as ANALYZER, INCAS, and JEWEL to provide profiles of utilization of various resources such as Disk Operations, CPU utilisation, Cache Performance etc. These tools even provide information about various kinds of overheads. Now, let us discuss various kinds of performance analysis tools.

2.6.1 Search-based Tools

The search-based tools firstly identify the problem and thereafter appropriately give advice on how to correct it.

AT Expert from Cray Research is one of the tools being used for enhancing the performance of Fortran programs with the help of a set of rules which have been written with Cray auto-tasking library. The Cray auto-tasking library assists in achieving the parallelism. Basically, the ATExpert analyses the Fortran program and tries to suggest complier directives that could help in improving the performance of the program.

Another tool called *"Performance Consultant"* is independent of any programming language, model and machines. It basically asks three questions i.e. WHY, WHERE and WHEN about the performance overheads and bottlenecks. These three questions form the 3 different axes of the hierarchal model. One of the important features of Performance Consultant tool is that it searches for bottlenecks during execution of program. The above-mentioned features assist in maintaining the reduced volume of data. The WHY axis presents the various bottlenecks such as communication, I/O etc. The WHERE axis defines the various sources which can cause bottlenecks such as Interconnection networks, CPU etc. The WHEN axis tries to separate the set of bottlenecks into a specific phase of execution of the program.

2.6.2 Visualisation

Visualization is a generic method in contract to search based tools. In this method visual aids are provided like pictures to assist the programmer in evaluating the performance of parallel programs. One of visualization tools is *Paragraph*. The Paragraph supports various kinds of displays like communication displays, task information displays, utilisation displays etc.

Communication Displays

Communication displays provide support in determining the frequency of communication, whether congestion in message queues or not, volume and the kind of patterns being communicated etc.

Communication Matrix

A communication matrix is constructed for displaying the communication pattern and size of messages being sent from one processor to another, the duration of communication etc. In the communication matrix, a two dimensional array is constructed such that both the horizontal and vertical sides are represented with the various processors. It mainly provides the communication pattern between the processors. The rows indicate the processor which is sending the message and columns indicate the processor which is going to receive the messages as shown in *Figure 12*.

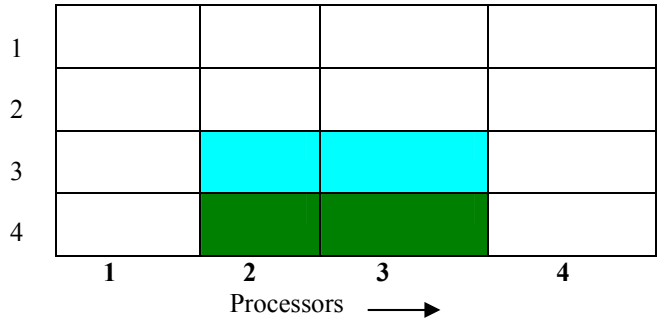


Figure: 12 Communication Matrix

Communication Traffic

The Communication Traffic provides a pictorial view of the communication traffic in the interconnection network with respect to the time in progress. The Communication Traffic displays the total number of messages, which have been sent but not received.

Message Queues

The message queue provides the information about the sizes of queues under utilization of various processors. It indicates the size of each processor incoming message queue, which would be varying depending upon the messages being sent, received or buffered as shown in *Figure 13*. It may be noted that every processor would be having a limit on maximum length of its queue. This display mainly helps in analyzing whether the communication is congestion free or not.

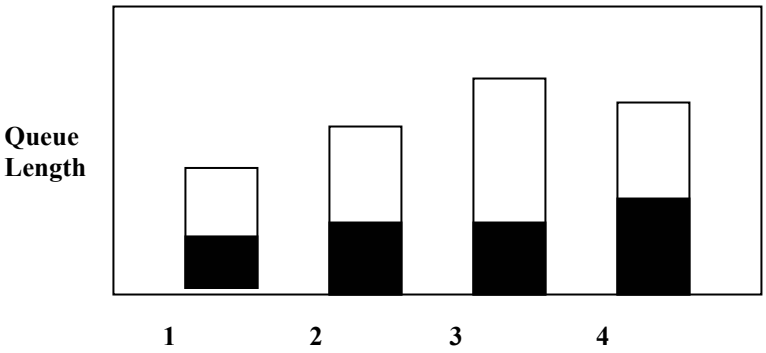


Figure 13: Message Queues

Processors Hypercube

This is specific to In the hypercube: Here, each processor is depicted by the set of nodes of the graph and the various arcs are represented with communication links. The nodes status can be idle, busy, sending, receiving etc. and are indicated with the help of a specific color. An arc is drawn between two processors when message is sent from one node to another and is deleted when the message is received.

Utilisation Displays

It displays the information about distribution of work among the processors and the effectiveness of the processors.

Utilisation Summary

The Utilisation Summary indicates the status of each processor i.e. how much time (in the form of percentage) has been spent by each processor in busy mode, overhead mode and idle mode as shown in *Figure 14*.

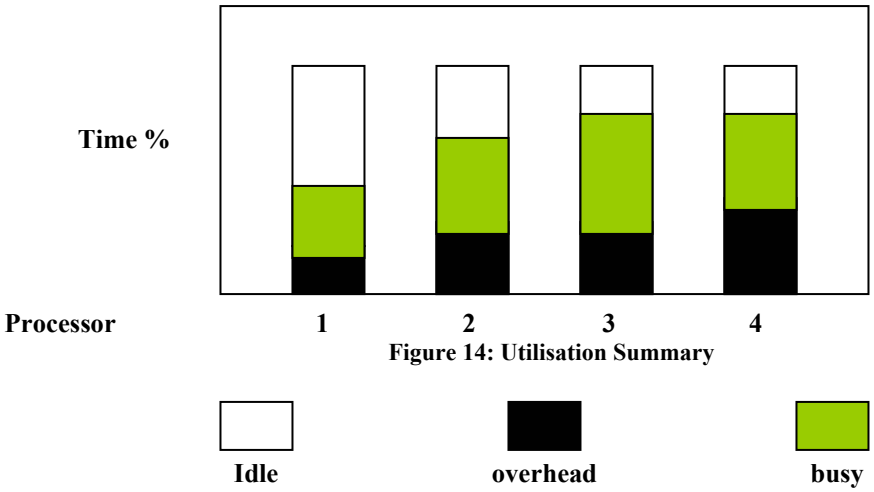


Figure 14: Utilisation Summary

Utilisation Count

The Utilisation Count displays the status of each processor in a specific mode i.e. busy mode, overhead mode and idle mode with respect to the progress in time as shown in *Figure 15*.

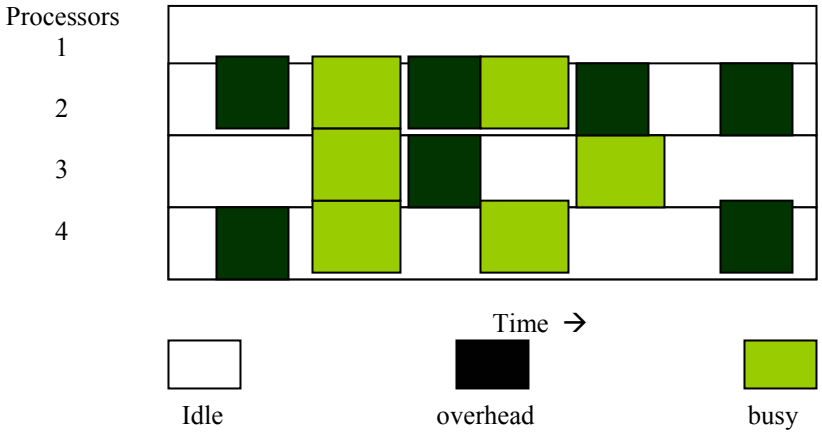


Figure 15: Utilisation Count

Gantt chart: The Gantt chart illustrates the various activities of each processor with respect to progress in time in idle-overhead -busy modes with respect to each processor.

Kiviat diagram: The Kiviat diagram displays the geometric description of each processor's utilization and the load being balanced for various processors.

Concurrency Profile: The Concurrency Profile displays the percentage of time spent by the various processors in a specific mode i.e. idle/overhead/busy.

Task Information Displays

The Task Information Displays mainly provide visualization of various locations in the parallel program where bottlenecks have arisen instead of simply illustrating the issues that can assist in detecting the bottlenecks. With the inputs provided by the users and through portable instrumented communication library i.e., PICL, the task information displays provide the exact portions of the parallel program which are under execution.

Task Gantt

The Task Gantt displays the various tasks being performed i.e., some kind of activities by the set of processors attached to the parallel computer as shown in *Figure 16.*

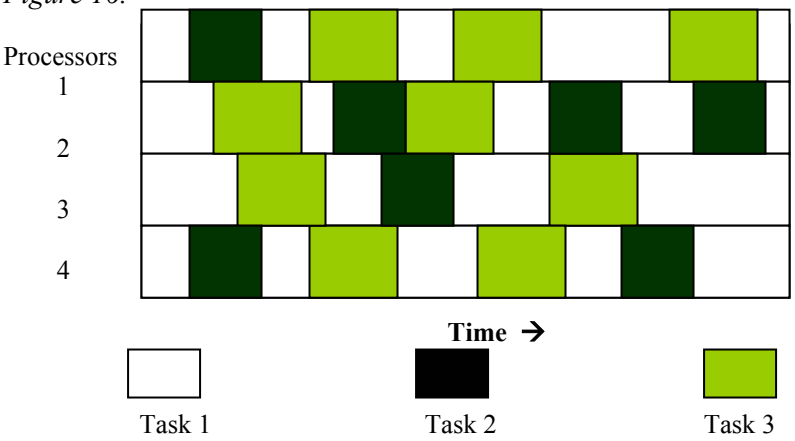


Figure 16: Task Gantt

Summary of Tasks

The Task Summary tries to display the amount of duration each task has spent starting from initialisation of the task till its completion on any processor as shown in *Figure 17.*

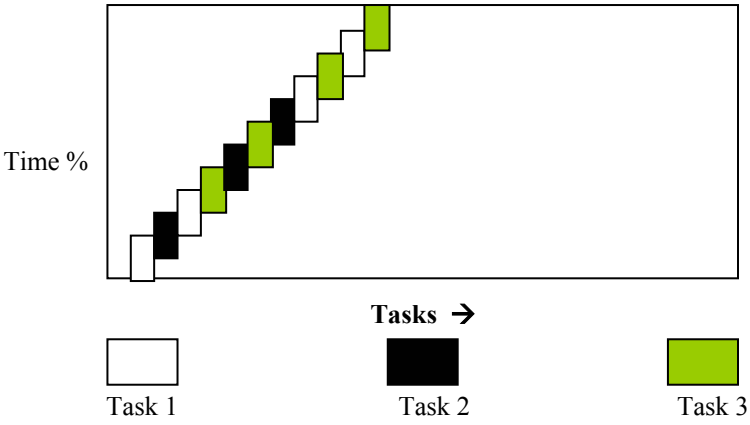


Figure 17: Summary of Tasks

2.7 PERFORMANCE INSTRUMENTATION

The performance instrumentation emphasizes on how to efficiently gather information about the computation of the parallel computer. The method of instrumentation mainly attempts to capture information about the applications by adding some kind of instrument like a code or a function or a procedure etc. in to the source code of the program or may be at the time of execution. The major effect of such instrumentation is generation of some useful data for performance tools. The performance instrumentation sometimes causes certain problems known as intrusion problems.

Check Your Progress 3

1) Which of the following are the parts of performance measurement?
   (a) Performance Analysis
   (b) Performance Instrumentation
   (c) Overheads
   (d) *A and B*

2) List the various search-based tools used in performance analysis.

……
……
……
……

3) List the various visualisation tools employed in performance analysis.

……
……
……
……

2.8 SUMMARY

The performance analysis of any parallel algorithm is dependent upon three major factors i.e., Time Complexity of the Algorithm, Total Number of Processors required and Total Cost Involved. However, it may be noted that the evaluation of performance in parallel computing is based on parallel computer in addition to the parallel algorithm for the various numerical as well non-numerical problems. The various kinds of performance metrics involved for analyzing the performance of parallel algorithms for parallel computers have been discussed e.g., Time Complexity, Speed-Up, Efficiency, MIPS, and Mflops etc. The speedup is directly proportional to number of processors; therefore a linear arc is depicted. However, in situations where there is parallel overhead in such states the arc is sub-linear. The efficiency of a program on a parallel computer with say N processors can be defined as the ratio of the relative speed up achieved while shifting from single processor machine to n processor machine to the number of processors being used for achieving the result in a parallel computer. The various sources of overhead in a parallel computer are; *Uneven load distribution, Cost involved in inter-processor communication, Synchronization, Parallel Balance Point etc.*

In this unit we have also discussed three speed-up laws i.e., Gustafson's law, Amdahl's law, Sun and Ni's law. The performance of the algorithms is one of the other major concerns. In order to analyze the performance of algorithms, there are various kinds of performance measurement tools. The measurement tools rely not only on the parallel algorithm but are also require to collect data from the operating system, the hardware being used for providing effective utilization of the tools. The task of measuring the performance of the parallel programs has been divided into two components i.e. Performance Analysis and Performance Instrumentation. The various tools employed by the above two components have also been discussed in this unit.

2.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) D
2) D
3) The running time defines the amount of time consumed in execution of an algorithm for a given input on the N processor based parallel computer. The running time is denoted by T(n) where n signifies the number of processors employed.
Speed up defines the ratio of the time required to execute a given program using a specific algorithm on a machine with single processor i.e. T (1) (where n=1) to the time required to execute a given program using a specific algorithm on a machine with multiple processor i.e. T(n).

Efficiency of the machine i.e. how are the resources of the machine e.g. processors being utilized (effective or ineffective)

Check Your Progress 2

1) C
2) A
3) B

Check Your Progress 3

1) D
2) ATExpert from Cray Research, Performance Consultant etc.
3) Utilisation Displays, Communication Displays and Task Information displays.

2.10 FURTHER READINGS

Parallel Computers - *Architecture and Programming*, Second edition, by V.Rajaraman and C.Siva Ram Murthy (Prentice Hall of India)

UNIT 3 RECENT TRENDS IN PARALLEL COMPUTING

Structure **Page Nos.**

3.0 INTRODUCTION

This unit discusses the current trends of hardware and software in parallel computing. Though the topics about various architectures, parallel program development and parallel operating systems have already been discussed in the earlier units, here some additional topics are discussed in this unit.

3.1 OBJECTIVES

After studying this unit you should be able to describe the

- various parallel programming models;
- concept of grid computing;
- concept of cluster computing;
- IA-64 architecture, and
- concept of Hyperthreading.

3.2 RECENT PARALLEL PROGRAMMING MODELS

A model for parallel programming is an abstraction and is machine architecture independent. A model can be implemented on various hardware and memory architectures. There are several parallel programming models like Shared Memory model, Threads model, Message Passing model, Data Parallel model and Hybrid model etc.

As these models are hardware independent, the models can (theoretically) be implemented on a number of different underlying types of hardware.

The decision about which model to use in an application is often a combination of a number of factors including the available resources and the nature of the application. There is no universally best implementation of a model, though there are certainly some implementations of a model better than others. Next, we discuss briefly some popular models.

1) Shared Memory Model

In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks / semaphores may be used to control access to the shared memory. An advantage of this model from the programmer's point of view is that program development can often be simplified. An important disadvantage of this model (in terms of performance) is that for this model, data management is difficult.

2) Threads Model

In this model a single process can have multiple, concurrent execution paths. The main program is scheduled to run by the native operating system. It loads and acquires all the necessary softwares and user resources to activate the process. A thread's work may best be described as a subroutine within the main program. Any thread can execute any one subroutine and at the same time it can execute other subroutine. Threads communicate with each other through global memory. This requires Synchronization constructs to insure that more than one thread is not updating the same global address at any time. Threads can be created and destroyed, but the main program remains live to provide the necessary shared resources until the application has completed. Threads are commonly associated with shared memory architectures and operating systems.

3) Message Passing Model

In the message-passing model, there exists a set of tasks that use their own local memories during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines. Tasks exchange data by sending and receiving messages. In this model, data transfer usually requires cooperation among the operations that are performed by each process. For example, a send operation must have a matching receive operation.

4) Data Parallel Model

In the data parallel model, most of the parallel work focuses on performing operations on a data set. The data set is typically organised into a common structure, such as an array or a cube. A set of tasks work collectively on the same data structure with each task working on a different portion of the same data structure. Tasks perform the same operation on their partition of work, for example, “add 3 to every array element” can be one task. In shared memory architectures, all tasks may have access to the data structure through the global memory. In the distributed memory architectures, the data structure is split up and data resides as “chunks” in the local memory of each task.

5) Hybrid model

The hybrid models are generally tailormade models suiting to specific applications. Actually these fall in the category of mixed models. Such type of application-oriented models keep cropping up. Other parallel programming models also exist, and will continue to evolve corresponding to new applications.

In this types of models, any two or more parallel programming models are combined. Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.

Another common example of a hybrid model is combining data parallel model with message passing model. As mentioned earlier in the data parallel model, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data transparently between tasks and the programmer.

6) Single Program Multiple Data (SPMD)

SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models. A single program is executed by all tasks simultaneously. SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program, they may execute only a portion of it. In this model, different tasks may use different data.

7) Multiple Program Multiple Data (MPMD)

Like SPMD, MPMD is actually a “high level” programming model that can be built upon any combination of the previously mentioned parallel programming models.
MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executed on the same or different program. In this model all tasks may use different data.

3.3 PARALLEL VIRTUAL MACHINE (PVM)

PVM is basically a simulation of a computer machine running parallel programs. It is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. It includes a combination of good features of various operating systems and architectures. Thus large computational problems can be solved more cost effectively by using the combined power and memory of many computers. The software is highly portable. The source, which is available free through netlib, has been compiled on a range of computing machines from laptops to CRAY machines.

PVM enables users to exploit their existing computer hardware to solve much larger problems at minimal additional cost. Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems. Further, PVM's are being used as an educational tools to teach parallel programming. With tens of thousands of users, PVM has become the de facto standard for distributed computing world-wide.

Check Your Progress 1

1) Explain the operations in the following message passing operating system model
   1) Object Oriented Model
   2) Node addressed model
   3) Channel addressed model

2) Explain various Multi Processor Execution Node.

   ……
   ……
   ……
   ……

3) What are the levels at which multitasking is exploited?

   ……
   ……
   ……
   ……

3.4 GRID COMPUTING

Grid Computing means applying the resources of many computers in a network simultaneously to a single problem for solving a scientific or a technical problem that requires a large number of computer processing cycles or accesses to large amounts of data. Grid computing uses software to divide and distribute pieces of a program to as many as several thousand computers. A number of corporations, professional groups and university consortia have developed frameworks and software for managing grid computing projects.

Thus, the Grid computing model allows companies to use a large number of computing resources on demand, irrespective of where they are located. Various computational tasks can be performed using a computational grid. Grid computing provides clustering of remotely distributed computing environment. The principal focus of grid computing to date has been on maximizing the use of available processor resources for compute-intensive applications. Grid computing along with storage virtualization and server virtualization enables a utility computing. Normally it has a Graphical User Interface (GUI), which is a program interface based on the graphics capabilities of the computer to create screens or windows.

Grid computing uses the resources of many separate computers connected by a network (usually the internet) to solve large-scale computation problems. The SETI@home project, launched in the mid-1990s, was the first widely-known grid computing project, and it has been followed by many others project covering tasks such as protein folding, research into drugs for cancer, mathematical problems, and climate models.

Grid computing offers a model for solving massive computational problems by making use of the unused resources (CPU cycles and/or disk storage) of large numbers of disparate, often desktop, computers treated as a virtual cluster embedded in a distributed telecommunications infrastructure. Grid computing focuses on the ability to support computation across administrative domains which sets it apart from traditional computer clusters or traditional distributed computing.

Various systems which can participate in the grid computing as platform are:
Windows 3.1, 95/98, NT, 2000 XP , DOS, OS/2, , supported by Intel (x86);
Mac OS A/UX (Unix) supported by Motorola 680 x0;
Mac OS , AIX (Unix), OS X (Unix) supported by Power PC;
HP / UX (Unix) supported by HP 9000 (PA – RISC);
Digital Unix open VMS, Windows NT spported by Compaq Alpha;
VMS Ultrix (Unix) supported by DEC VAX;
Solaris (Unix) supported by SPARC station;
AIX (Unix) supported by IBM RS / 6000;
IRIS (Unix) supported by Silicon Graphics workstation.

3.5 CLUSTER COMPUTING

The concept of clustering is defined as the use of multiple computers, typically PCs or UNIX workstations, multiple storage devices, and their interconnections, to form what appears to users as a single highly available system. Workstation clusters is a collection of loosely-connected processors, where each workstation acts as an autonomous and independent agent. The cluster operates faster than normal systems.

In general, a 4-CPU cluster is about 250~300% faster than a single CPU PC. Besides, it not only reduces computational time, but also allows the simulations of much bigger computational systems models than before. Because of cluster computing overnight

analysis of a complete 3D injection molding simulation for an extremely complicated model is possible.
Cluster workstation defined in ***Silicon Graphics*** project is as follows:

"A distributed workstation cluster should be viewed as single computing resource and not as a group of individual workstations".

The details of the cluster were: 150MHz R4400 workstations, 64MB memory, 1 GB disc per workstation, 6 x 17" monitors, Cluster operating on local 10baseT Ethernet, Computing Environment Status, PVM, MPI (LAM and Chimp), Oxford BSP Library.

The operating system structure makes it difficult to exploit the characteristics of current clusters, such as low-latency communication, huge primary memories, and high operating speed. Advances in network and processor technology have greatly changed the communication and computational power of local-area workstation clusters. Cluster computing can be used for load balancing in multi-process systems

A common use of cluster computing is to balance traffic load on high-traffic Web sites. In web operation a web page request is sent to a manager server, which then determines which of the several identical or very similar web servers to forward the request to for handling. The use of cluster computing makes this web traffic uniform.

Clustering has been available since the 1980s when it was used in DEC's VMS systems. IBM's SYSLEX is a cluster approach for a mainframe system. Microsoft, Sun Microsystems, and other leading hardware and software companies offer clustering packages for scalability as well as availability. As traffic or availability assurance increases, all or some parts of the cluster can be increased in size or number. Cluster computing can also be used as a relatively low-cost form of parallel processing for scientific and other applications that lend themselves to parallel operations

An early and well-known example was the project in which a number of off-the-shelf PCs were used to form a cluster for scientific applications.

Windows cluster is supported by the Symmetric Multiple Processors (SMP) and by Moldex3D R7.1. While the users chose to start the parallel computing, the program will automatically partition one huge model into several individual domains and distribute them over different CPUs for computing. Every computer node will exchange each other's computing data via message passing interface to get the final full model solution. The computing task is parallel and processed simultaneously. Therefore, the overall computing time will be reduced greatly.

Some famous projects on cluster computing are as follows:

i) **High Net Worth Project**: (developed by: Bill McMillan, JISC NTI/65 – The HNW Project, University of Glasgow, (billm@aero.gla.ac.uk)

The primary aims / details of the project are:

- Creation of a High Performance Computing Environment using clustered workstations
- Use of pilot facility,
- Use of spare capacity,
- Use of cluster computing environment as a Resource Management system,
- Use of ATM communications,
- Parallel usage between sites across SuperJANET,
- Promoting Awareness of Project in Community.

ii) **The primary aims/details of Load Sharing Facility Resource Management Software(LSFRMS)** are better resource utilisation by routing the task to the most appropriate system and better utilisation of busy workstations through load sharing, and also by involving idle workstations.

Types of Users of the LSFRMS:

- users whose computational requirements exceed the capabilities of their own hardware;
- users whose computational requirements are too great to be satisfied by a single workstation;
- users intending to develop code for use on a national resource;
- users who are using national resources
- users who wish to investigate less probable research scenarios.

Advantages of using clusters:

- Better resource utilisation;
- Reduced turnaround time;
- Balanced loads;
- Exploitation of more powerful hosts;
- Access to non-local resources;
- Parallel and distributed applications.

Check Your Progress 2

1) Name any three platforms which can participate in grid computing.
……
……
……
……

2) Explain the memory organization with a cluster computing.
……
……
……
……

3) Find out the names of famous cluster computer projects.
……
……
……
……

4) Explain various generations of message passing multi computers.
……
……
……
……

3.6 INTEL ARCHITECTURE – 64 (IA-64)

IA-64 (Intel Architecture-64) is a 64-bit processor architecture developed in cooperation by Intel and Hewlett-Packard, implemented by processors such as Itanium. The goal of Itanium was to produce a "post-RISC era" architecture using EPIC(Explicitly Parallel Instruction Computing).

1 EPIC Architecture

In this system a complex decoder system examines each instruction as it flows through the pipeline and sees which can be fed off to operate in parallel across the available execution units — *e.g.*, a sequence of instructions for performing the computations
A = B + C and
D = F + G

These will be independent of each other and will not affect each other, and so they can be fed into two different execution units and run in parallel. The ability to extract instruction level parallelism (ILP) from the instruction stream is essential for good performance in a modern CPU.

Predicting which code can and cannot be split up this way is a very complex task. In many cases the inputs to one line are dependent on the output from another, but only if some other condition is true. For instance, consider the slight modification of the example noted before, A = B + C; IF A==5 THEN D = F + G. In this case the calculations remain independent of the other, but the second command requires the results from the first calculation in order to know if it should be run at all.

In these cases the circuitry on the CPU typically "guesses" what the condition will be. In something like 90% of all cases, an IF will be taken, suggesting that in our example the second half of the command can be safely fed into another core. However, getting the guess wrong can cause a significant performance hit when the result has to be thrown out and the CPU waits for the results of the "right" command to be calculated. Much of the improving performance of modern CPUs is due to better prediction logic, but lately the improvements have begun to slow. Branch prediction accuracy has reached figures in excess of 98% in recent Intel architectures, and increasing this figure can only be achieved by devoting more CPU die space to the branch predictor, a self-defeating tactic because it would make the CPU more expensive to manufacture.

IA-64 instead relies on the compiler for this task. Even before the program is fed into the CPU, the compiler examines the code and makes the same sorts of decisions that would otherwise happen at "run time" on the chip itself. Once it has decided what paths to take, it gathers up the instructions it knows can be run in parallel, bundles them into one larger instruction, and then stores it in that form in the program.

Moving this task from the CPU to the compiler has several advantages. First, the compiler can spend considerably more time examining the code, a benefit the chip itself doesn't have because it has to complete the task as quickly as possible. Thus the compiler version can be considerably more accurate than the same code run on the chip's circuitry. Second, the prediction circuitry is quite complex, and offloading a prediction to the compiler reduces that complexity enormously. It no longer has to examine anything; it simply breaks the instruction apart again and feeds the pieces off to the cores. Third, doing the prediction in the compiler is a one-off cost, rather than one incurred every time the program is run.

The downside is that a program's runtime-behaviour is not always obvious in the code used to generate it, and may vary considerably depending on the actual data being processed. The out-of-order processing logic of a mainstream CPU can make decisions on the basis of actual run-time data which the compiler can only guess at. It means that it is possible for the compiler to get its prediction wrong more often than comparable (or simpler) logic placed on the CPU. Thus this design this relies heavily on the performance of the compilers. It leads to decrease in microprocessor hardware complexity by increasing compiler software complexity.

Registers: The IA-64 architecture includes a very generous set of registers. It has an 82-bit floating point and 64-bit integer registers. In addition to these registers, IA-64 adds in a register rotation mechanism that is controlled by the Register Stack Engine. Rather than the typical spill/fill or window mechanisms used in other processors, the Itanium can rotate in a set of new registers to accommodate new function parameters or temporaries. The register rotation mechanism combined with predication is also very effective in executing automatically unrolled loops.

Instruction set: The architecture provides instructions for multimedia operations and floating point operations.

The Itanium supports several bundle mappings to allow for more instruction mixing possibilities, which include a balance between serial and parallel execution modes. There was room left in the initial bundle encodings to add more mappings in future versions of IA-64. In addition, the Itanium has individually settable predicate registers to issue a kind of runtime-determined "cancel this command" directive to the respective instruction. This is sometimes more efficient than branching.

Pre-OS and runtime sub-OS functionality

In a raw Itanium, a "Processor Abstraction Layer" (PAL) is integrated into the system. When it is booted the PAL is loaded into the CPU and provides a low-level interface that abstracts some instructions and provides a mechanism for processor updates distributed via a BIOS update.
During BIOS initialization an additional layer of code, the "System Abstraction Layer" (SAL) is loaded that provides a uniform API for implementation-specific platform functions.

On top of the PAL/SAL interface sits the "Extensible Firmware Interface" (EFI). EFI is not part of the IA-64 architecture but by convention it is required on all IA-64 systems. It is a simple API for access to logical aspects of the system (storage, display, keyboard, etc) combined with a lightweight runtime environment (similar to DOS) that allows basic system administration tasks such as flashing BIOS, configuring storage adapters, and running an OS boot-loader.

Once the OS has been booted, some aspects of the PAL/SAL/EFI stack remain resident in memory and can be accessed by the OS to perform low-level tasks that are implementation-dependent on the underlying hardware.

IA-32 support

In order to support IA-32, the Itanium can switch into 32-bit mode with special jump escape instructions. The IA-32 instructions have been mapped to the Itanium's functional units. However, since, the Itanium is built primarily for speed of its EPIC-style instructions, and because it has no out-of-order execution capabilities, IA-32 code executes at a severe performance penalty compared to either the IA-64 mode or the Pentium line of processors. For example, the Itanium functional units do not automatically generate integer flags as a side effect of ordinary ALU computation, and do not intrinsically support multiple outstanding unaligned memory loads. There are also IA-32 software emulators which are freely available for Windows and Linux, and these emulators typically outperform the hardware-based emulation by around 50%. The Windows emulator is available from Microsoft, the Linux emulator is available from some Linux vendors such as Novell and from Intel itself. Given the superior performance of the software emulator, and despite the fact that IA-32 hardware accounts for less than 1% of the transistors of an Itanium 2, Intel plan to remove the circuitry from the next-generation Itanium 2 chip codenamed "Montecito".

3.7 HYPER-THREADING

Hyper-threading, officially called **Hyper-threading Technology** (**HTT**), is Intel's trademark for their implementation of the simultaneous multithreading technology on the Pentium 4 microarchitecture. It is basically a more advanced form of Super-threading that was first introduced on the Intel Xeon processors and was later added to Pentium 4 processors. The technology improves processor performance under certain workloads by providing useful work for execution units that would otherwise be idle for example during a cache miss.

Features of Hyper-threading

The salient features of hyperthrading are:

i) Improved support for multi-threaded code, allowing multiple threads to run simultaneously.

ii) Improved reaction and response time, and increased number of users a server can support.

According to Intel, the first implementation only used an additional 5% of the die area over the “normal” processor, yet yielded performance improvements of 15-30%.
Intel claims up to a 30% speed improvement with respect to otherwise identical, non-SMT Pentium 4. However, the performance improvement is very application dependent, and some programs actually slow down slightly when HTT is turned on.

This is because of the replay system of the Pentium 4 tying up valuable execution resources, thereby starving for the other thread. However, any performance degradation is unique to the Pentium 4 (due to various architectural nuances), and is not characteristic of simultaneous multithreading in general.

Functionality of hypethread Processor

Hyper-threading works by duplicating those sections of processor that store the architectural state—but not duplicating the main execution resources. This allows a Hyper-threading equipped processor to pretend to be two “logical” processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously. Where execution resources in a non-Hyper-threading capable processor are not used by the current task, and especially when the processor is stalled, a Hyper-threading equipped processor may use those execution resources to execute the other scheduled task.

Except for its performance implications, this innovation is transparent to operating systems and programs. All that is required to take advantage of Hyper-Threading is symmetric multiprocessing (SMP) support in the operating system, as the logical processors appear as standard separate processors.

However, it is possible to optimize operating system behaviour on Hyper-threading capable systems, such as the Linux techniques discussed in Kernel Traffic. For example, consider an SMP system with two physical processors that are both Hyper-Threaded (for a total of four logical processors). If the operating system's process scheduler is unaware of Hyper-threading, it would treat all four processors similarly.

As a result, if only two processes are eligible to run, it might choose to schedule those processes on the two logical processors that happen to belong to one of the physical processors. Thus, one CPU would be extremely busy while the other CPU would be

completely idle, leading to poor overall performance. This problem can be avoided by improving the scheduler to treat logical processors differently from physical processors; in a sense, this is a limited form of the scheduler changes that are required for NUMA systems.

The Future of Hyperthreading

Current Pentium 4 based MPUs use Hyper-threading, but the next-generation cores, Merom, Conroe and Woodcrest will not. While some have alleged that this is because Hyper-threading is somehow energy inefficient, this is not the case. Hyper-threading is a particular form of multithreading, and multithreading is definitely on Intel roadmaps for the generation after Merom/Conroe/Woodcrest. Quite a few other low power chips use multithreading, including the PPE from the Cell processor, the CPUs in the Playstation 3 and Sun's Niagara. Regarding the future of multithreading
the real question is not whether Hyper-threading will return, as it will, but rather how it will work. Currently, Hyper-threading is identical to Simultaneous Multi-Threading, but future variants may be different. In future trends parallel codes have been easily ported to the Pilot Cluster, often with improved results and Public Domain and Commercial Resource Management Systems have been evaluated in the Pilot Cluster Environment.

The proposed enhancements in these architecture are as follows:

- High Performance language implementation
- Further developments in Heterogeneous Systems
- Management of wider domains with collaborating departments
- Faster communications
- Inter-campus computing

Check Your Progress 3

1) How is performance enhanced in IA-64 Architecture?
   ……
   ……
   ……
   ……
2) How is performance judged on the basis of run time behaviour?
   ……
   ……
   ……
   ……
   ……
3) List some data parallelism feature of a modern computer architecture.
   ……
   ……
   ……
   ……

3.8 SUMMARY

This unit discusses basic issues in respect of latest trends of hardware and software technology for parallel computing systems. The technology of parallel computing is the outcome of four decades of research and industrial advances in various fields like microelectronics, integration technologies, advanced processors, memory and peripheral devices, programming language development, operating system developments etc.

The high performance computers which are developed using these concepts provide fast and accurate solutions to scientific, engineering, business, social and aerospace applications.

The unit discusses some of the latest architectures. However, the learner is advised to get the knowledge about further ongoing developments from related websites.

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) i) **Object Oriented Model:** This model treats multiple tasks or processor concurrently, each task using its own address space. In this model multitasking is practiced at each node.
The communication between different tasks is carried out by either at synchronous or asynchronous message passing protocol. Message passing supports IPC as well as process migration or load balancing.

ii) **Node Address Model:** In this only one task runs on each node at a time. It is implemented with asynchronous scheme. The actual topology is hidden from the user. Some models use store and forward message routing and some models use wormhole routing.

iii) **Channel Addressed Model:** This model runs one task on each node. Data are communicated by synchronous message passing for communication channel.

2) a) Multi Processor super computers are built in vector processing as well as for parallel processing across multiple processors. Various executions models are parallel execution from fine grain process level.
b) Parallel execution from coarse grain process level
c) Parallel execution to coarse grain exam level

3) Multi tasking exploits parallelism at the following level:

i) The different functional unit is pipeline together.
ii) Various functional units are used concurrently.
iii) I/O and CPU activities are overlapped.
iv) Multiple CPUs can operate on a single program to achieve minimum execution time.

In a multi tasking environment, the task and the corresponding data structure of a program are properly pertaining to parallel execution in a such a manner that conflicts will not occur.

Check Your Progress 2

1) The following platform can participate in grid computing.
i) Digital, Unit, Open, VMS
ii) AIX supported by IBM RS/6000.
iii) Solaris (Unix supported by SPARC station)

2) In cluster computer the processor are divided into several clusters. Each cluster is a UMA or NUMA multiprocessor. The clusters are connected to global shared memory model. The complete system works on NUMA model.

All processing elements belonging to it clusters are allowed to access the cluster shared memory modules.

All clusters have access to global memory. Different computers systems have specified different access right among Internet cluster memory. For example, CEDAR multi processor built at University of Illinois adopts architecture in each cluster is Alliant FX/80 multiprocessor.

3) Collect them from the website.

4) Multi computers have gone through the following generations of development. The first generally ranged from 1983-87. It was passed on Processor Board Technology using Hypercube architecture and Software controlled message switching. Examples of this generation computers are Caltech, Cosmic and Intel, PEPSC.

The second generation ranged from 1988-92. It was implemented with mesh connected architecture, hardware access routing environment in medium grain distributed computing. Examples of such systems are Intel Paragon and Parsys super node 1000.

The third generation ranged from 1983-97 and it is an era of fine grain computers like MIT, J Machine and Caltech mosaic.

Check Your Progress 3

1) The performance in IA 64 architecture is enhanced in a modern CPU due to vector prediction logic and in branch prediction technique the path which is expected to be taken as is anticipated in advance, in order to preset the required instruction in advance. Branch prediction accuracies has reached to more than 98% in recent Intel architecture, and such a high figure is achieved by devoting more CPU die space to branch prediction.

2) The run time behaviour of a program can be adjudged only at execution time. It depends on the actual data being processed. The out of order processing logic of a main stream CPU can make itself on the basis of actual run time data which only compiler can connect.

3) Various data parallelism depends as specified as to how data is access to distribute in SIMD or MIMD computer. These are as follows:

   i) **Run time automatic decomposition:** In this data are automatically distributed with no user intervention.

   ii) **Mapping specifications:** It provides the facility for user to specify the communication pattern.

   iii) **Virtual processor support:** The compiler maps virtual process dynamically and statically.

   iv) **Direct Access to sharing data:** Shared data can be directly accessed without monitor control.

3.10 FURTHER READINGS

1) Thomas L. Casavant, Pavel Tvrdik, Frantisck Plasil, *Parallel Computers: Theory and Applications*, IEEE Computer Society Press (1996)

2) Kai Hwang: *Advanced Computer Architecture: Parallelism*, Scalability and Programmability, Tata-McGraw-Hill (2001)