Q.1. Discuss each of the following concepts, with at least one appropriate example not discussed in course material.

i) Temporal Parallelism

ii) Thread

***iii)* Data-flow computing**

iv) Data Flow Computing

Ans:-

i) 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.

ii) 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.

The simple example shown in full on the previous page defines two classes:

```
class SimpleThread extends Thread {
   public SimpleThread(String str) {
           super(str);
   }
   public void run() {
           for (int i = 0; i < 10; i++) {
              System.out.println(i + " " + getName());
        try {
                   sleep((int)(Math.random() * 1000));
              } catch (InterruptedException e) {}
           }
           System.out.println("DONE! " + getName());
   }
}
```

iii) Data-flow 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.

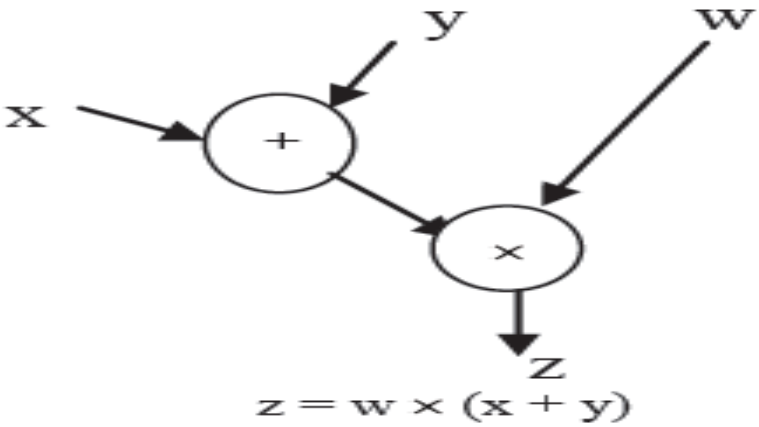


Figure 2: DFG for z = w × (x+y)

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.

IV) Data-flow 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.

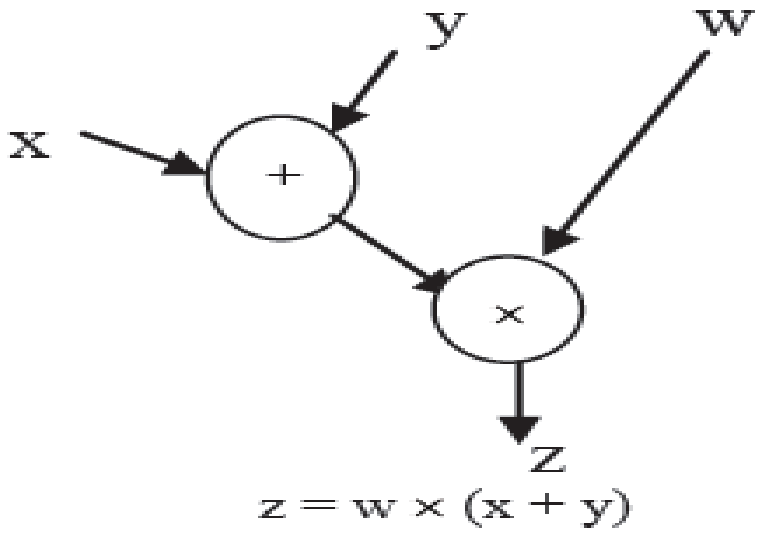


Figure 2: DFG for z = w × (x+y)

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.

Q.2. a) Draw the dataflow graph for the sequence of instructions z = w × (x + y); u = z × v

Ans:-The dataflow graph for the instruction z = w × (x+y) is shown in *Figure 2*.

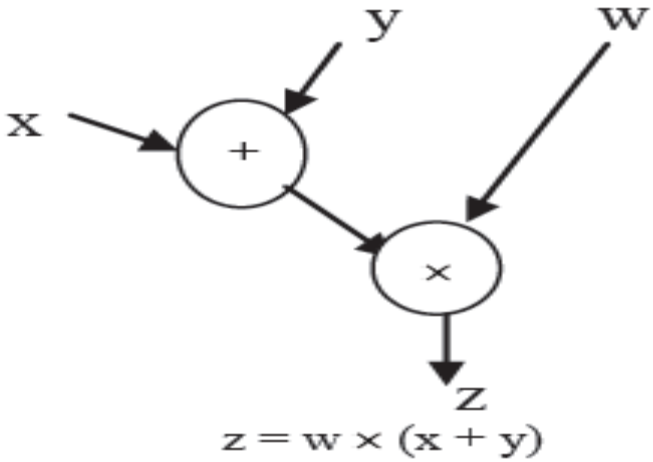
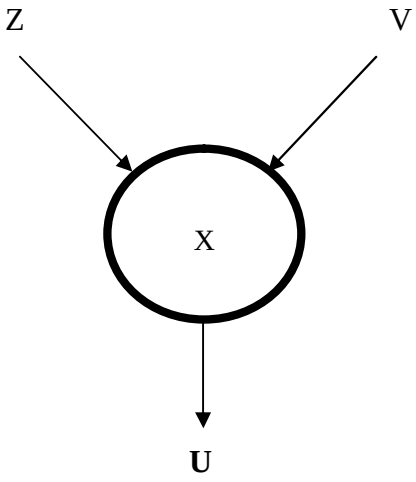


Figure 2: DFG for z = w × (x+y)

The dataflow graph for the instruction



***b)* Discuss essential features of each of the following scheme for classification of parallel computers:**
(i) Handler's
(ii) Uniform Memory Access Model
(iii) Non-Uniform Memory Access Mode l
(iv) Cache-only Memory Architecture Model

Ans:- (i) Handler's: In 1977, Wolfgang Handler proposed an elaborate notation for expressing the pipelining and parallelism of computers. Handler's classification is best explained by showing how the rules and operators are used to classify several machines.

- 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 onebit operations in the ALU.

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.

(ii) Uniform Memory Access Model: 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.

(iii) Non-Uniform Memory Access Model : 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.

(iv) Cache-only Memory Architecture Model : 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.

c) Use Bernstein's conditions for determining the maximum parallelism between the instructions in the following segment.

S1: N = 2 × X + 3× Z
S2: R = 4 ×U + X
S3: S = 3 ×Z + 2 ×V
S4: Z = 5 ×Y + Z
S5: P = 2 ×Y + R

Ans:- 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 RI that consists of memory locations read by the statement of instruction I1.
ii) The Write set or output set WI that consists of memory locations written into by instruction I1.

The sets RI and WI are not disjoint as the same locations are used for reading and writing by SI.
The following are Bernstein Parallelism conditions which are used to determine whether statements are parallel or not:

1) Locations in R1 from which S1 reads and the locations W2 onto which S2 writes must be mutually exclusive. That means S1 does not read from any memory location onto which S2 writes. It can be denoted as:
R1 W2=Φ

2) Similarly, locations in R2 from which S2 reads and the locations W1 onto which S1 writes must be mutually exclusive. That means S2 does not read from any memory location onto which S1 writes. It can be denoted as:
R2 W1=Φ

3) The memory locations W1 and W2 onto which S1 and S2 write, should not be read by S1 and S2. That means R1 and R2 should be independent of W1 and W2. It can be denoted as :
W1 W2=Φ

Q.3. *a)* Discuss Permutation Networks in detail

Ans:- 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 $1 \leftrightarrow 5$, $2 \leftrightarrow 4$, $3 \leftrightarrow 1$, $4 \leftrightarrow 3$, $5 \leftrightarrow 2$.

The connections are shown in the *Figure 13*.

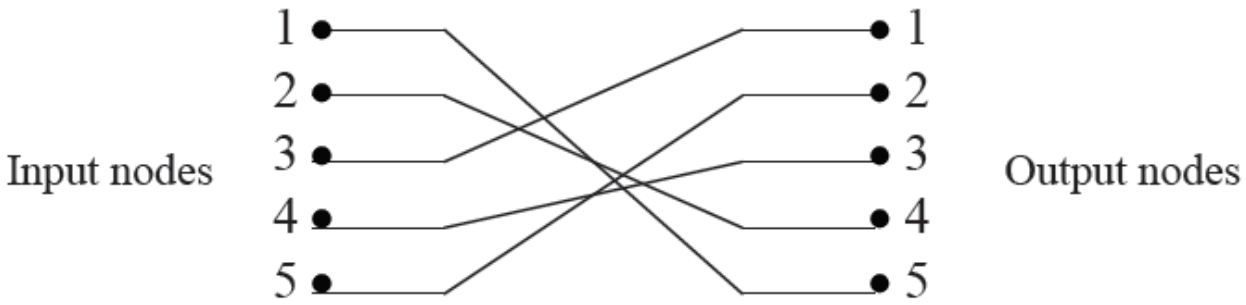


Figure 13: Node-Connections

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 Xn-1, Xn-2, X0 (N is chosen such that N = 2n.) The perfect shuffle of these N objects is expressed as
Xn-1, Xn-2, X0 = Xn-2, X0 Xn-1.
2) Butterfly permutation: This permutation is obtained by interchanging the most significant bit in address with least significant bit.
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.

***b)* Discuss relative merits and demerits of Cross-bar Interconnection Network and Systolic Array Network.**

Ans:- Cross-bar Interconnection Network: 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)*.

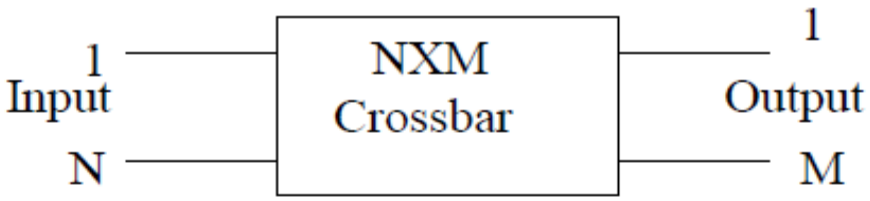


Figure: 3(a)

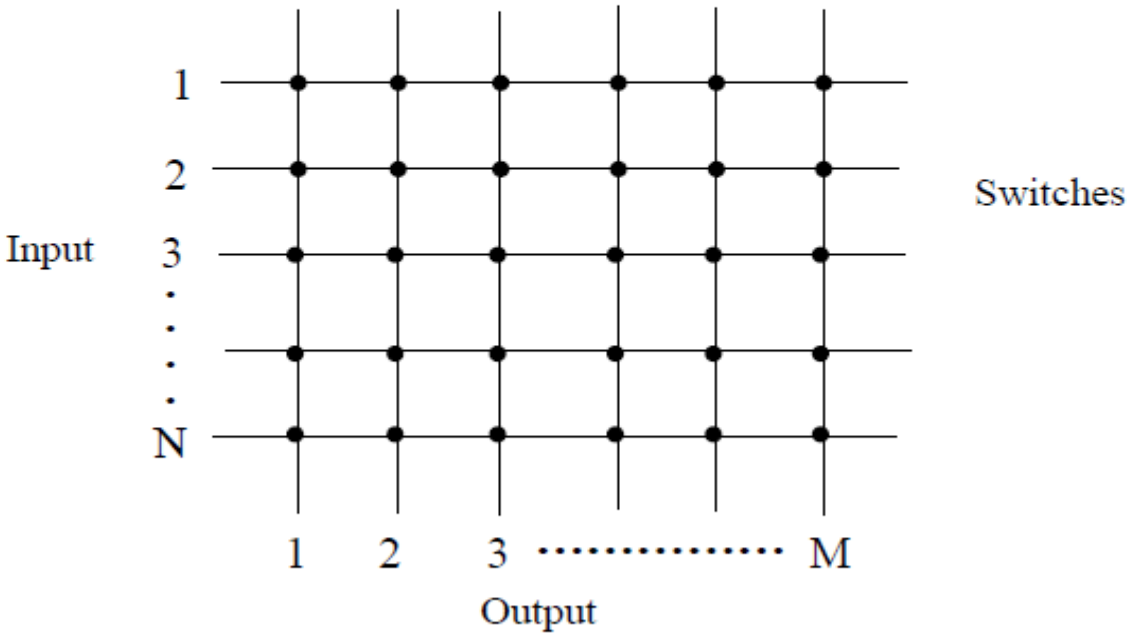


Figure: 3(b)
Figure 3: Crossbar Network

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 N2. It creates delay equivalent to one switching operation and the routing control mechanism is easy. The crossbar network requires N2 switches for N input and N output network.

Systolic Array Network: 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.

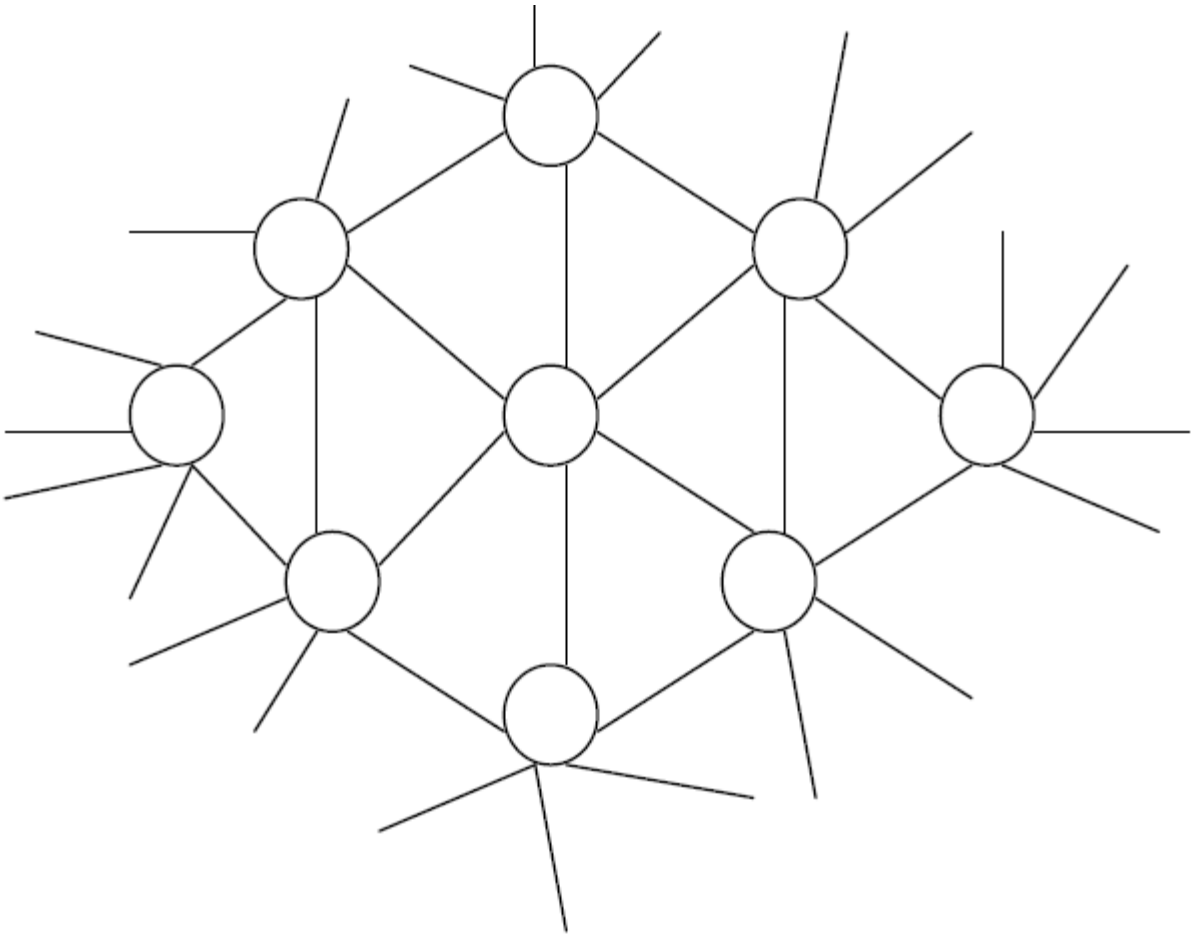


Figure 10: Systolic Array

c) For K-ary n-cube network calculate each of the following

i) Number of nodes in the network

ii) The Network diameter

iii) Bisection bandwidth of the network.

Ans:- i) Number of nodes in the network: In a k-ary n-cube network, the number of nodes N= kn for the torus and N = 2n for the hypercube. In the dual network, the number of nodes N' = n' k' n' for the torus and N' = n' 2 n'-1 for the hypercube. To have the same number of nodes N = N' in the k-ary n-cube network and its dual:

$K^n = n' k'^{n'}$ for the torus, and — Equation 1

$2^n = n' 2^{n'-1}$ for the hypercube — Equation 2

Solving Equation 1 with n = n' gives:

$$\frac{k}{k'} = \sqrt[n]{n}$$ — Equation 3

Hence, there are more nodes per dimension in a torus than that of its dual with equation dimensionality and number of nodes.
Solving Equation 2 yields:
n = n' +log2(n'/2) Equation 4
Thus the dimensionality of a hypercube is greater than of its dual with an equal number of nodes. For example, there are 1024 nodes in a 10-D hypercube, as well as in the dual hypercube with 8 dimensions.

ii) The 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.

In other words, the network diameter is defined as the maximum distance between any two nodes in the network. It is calculated by counting the number of hops between the two most distance nodes in the network.
In k-ary n-cube network, the diameter D= nk/2 for a torus, and D= n for a hypercube, in the dual network the diameter D' = n' k' / 2 for a torus, and D' = n' for a hypercube.
If the dimensionalities of a torus and its dual are equal, n = n', and for an equal number of nodes:

$$\frac{D}{D\prime} = \frac{k}{k\prime} = \sqrt[n]{n}$$

For a hypercube:

$$\frac{D}{D'} = \frac{n}{n'} = \frac{n' + \log_2(n'/2)}{n'}$$

Hence, the diameter of a k-ary n-cube network is larger than that of its dual with an equal number of nodes.

iii) Bisection bandwidth of the network: 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.

In other words , The bisection width of the network is defined as the number of channels that must be crossed in order to cut the network into two equal sub-networks. The bisection width of a k-ary n-cube networks torus is b = 2k n-1. The factor 2 is due to the ring arrangement in all dimensions. The bisection width of a hypercube is b = 2kn-1. In the dual network, the bisection width is b' = 2k'n'-1 for a torus, and b' = 2n'-1 for a hypercube.
For a torus and its dual network with an equal number of nodes and same dimensionality:

$$\frac{b}{b\prime} = \frac{k^{n-1}}{k\prime^{n-1}} = \frac{k^n}{k\prime^n} X \frac{k\prime}{k} = \frac{n}{\sqrt[n]{n}}$$

For a hypercube network and its dual with an equal number of nodes:

$$\frac{b}{b'} = \frac{2^{n-1}}{2'^{n-1}} = \frac{n'}{k2}$$

Therefore, the bisection width of a k-ary n-cube network is larger than that of its dual with an equal number of nodes.

Q.4. Write brief notes on the following:

i) Array processing
ii) Associative Array Processing
iii) VLIW architecture
iv) Multi-threaded processor

Ans:- i) Array processing : Array processing is another method of vector processing. 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 of Array processing is same as in SIMD. An array processor can handle one instruction and multiple data streams as same in case of SIMD organisation. Therefore, array processors are also called SIMD array computers.

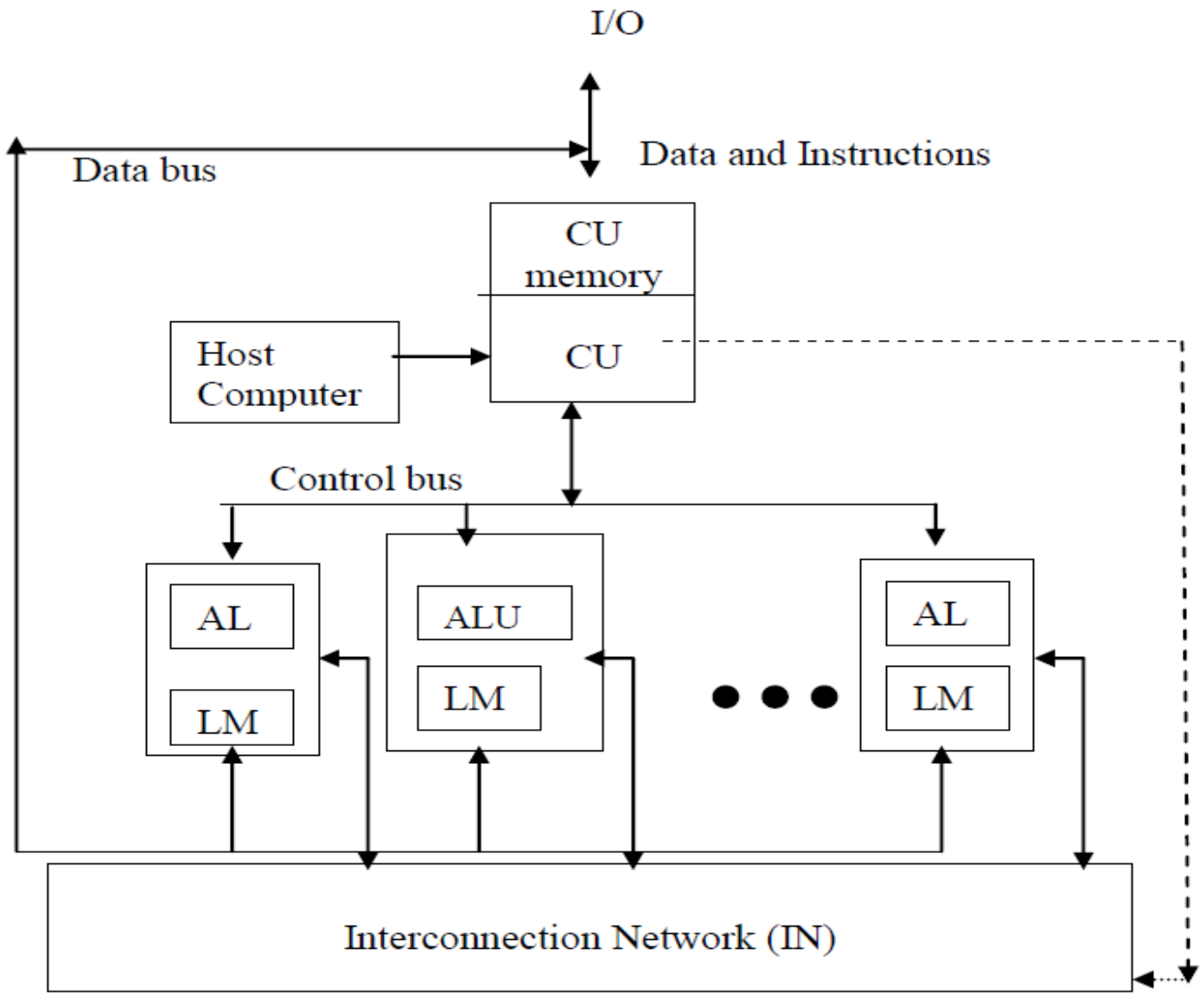


Figure 7: Organisation of SIMD Array Processor

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.
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.

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.

ii) Associative Array Processing: The Array processor built with associative memory is called Associative Array Processor. 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.

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 andparallel 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 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.

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.

iii) VLIW architecture : 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.

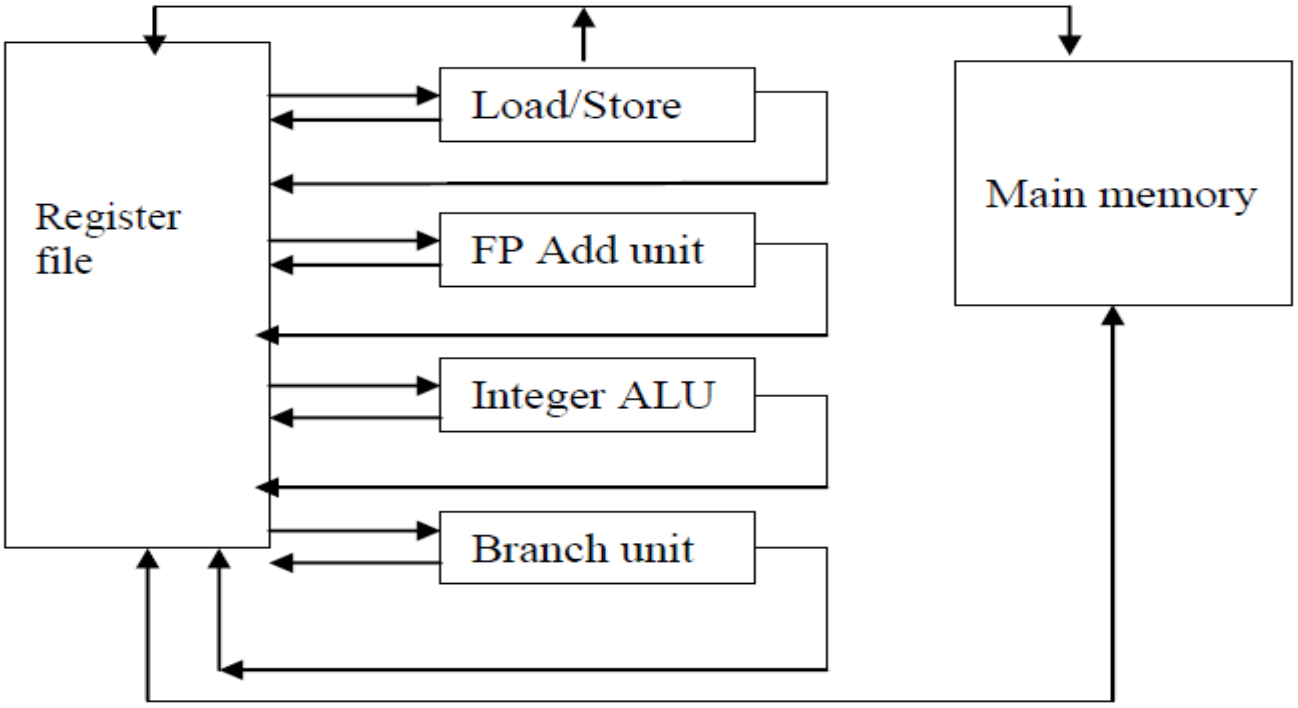


Figure 11: VLIW Processor

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.

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.

iv) Multi-threaded processor: 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). In case of large-scale MPP systems, the following two problems arise:

These problems increase in the design of large-scalemultiprocessors 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 case of large-scale MPP systems, the following two problems arise:

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.

For example, one processor in a multiprocessor system needs two memory loads of two variables from two remote processors as shown in *Figure 12*.

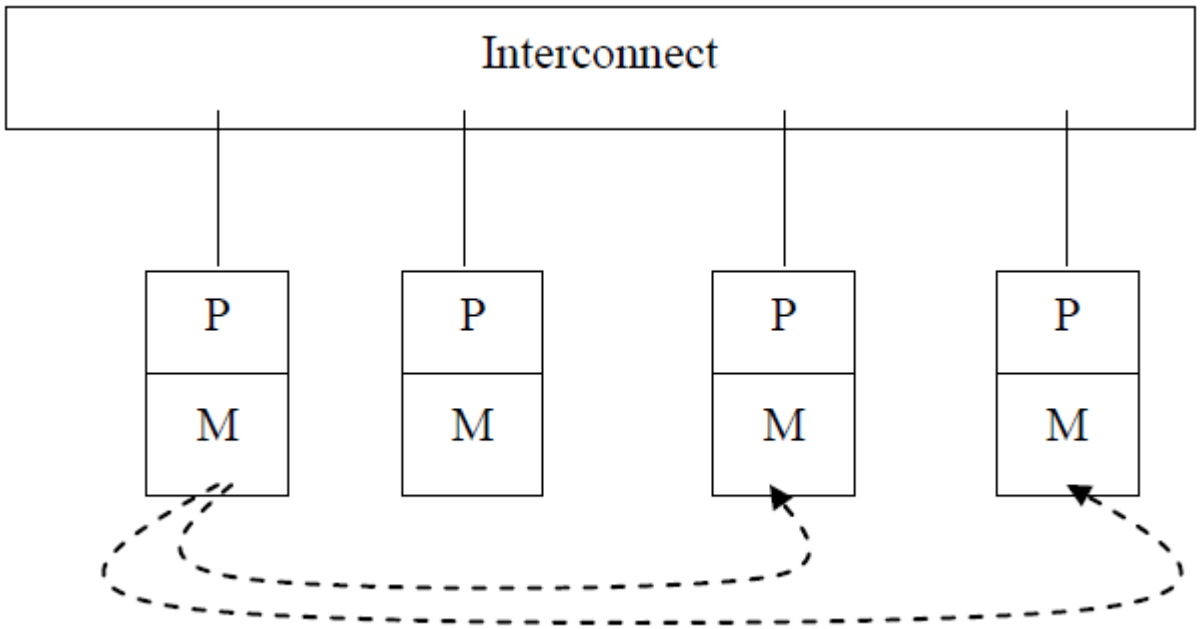


Figure 12: Latency problems in MPP

Q.5. a)Using sorting algorithm for combinational circuit given in Section 1.7 of Block 2, sort the following sequence of numbers in increasing order. 18, 15, 13, 10, 9, 12, 20, 14, 1000, 50, 1055, 0, 70, 33, 93
Ans:-

b) Using matrix multiplication algorithm given in Section 1.10, compute the following product:

$$\begin{pmatrix} 12 & 31 \\ 32 & 16 \end{pmatrix} \begin{pmatrix} 5 & 14 \\ 7 & 22 \end{pmatrix}$$

Ans:- 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:
Oij = Summation x of (M1ix * M2xj) x=1 to b, where 1<i<a and 1<j<c.

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
```

For the given matrixe:

$$\begin{pmatrix} 12 & 31 \\ 32 & 16 \end{pmatrix} \begin{pmatrix} 5 & 14 \\ 7 & 22 \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{60+217} & \mathbf{168+682} \\ \mathbf{160+112} & \mathbf{448+352} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{277} & \mathbf{850} \\ \mathbf{272} & \mathbf{800} \end{pmatrix}$$

Q.6. *a)* Discuss Odd-Even Merging Circuit for sorting and then analyse for its complexity.

Ans:- MERGE SORT CIRCUIT: 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.

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 thu 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*.

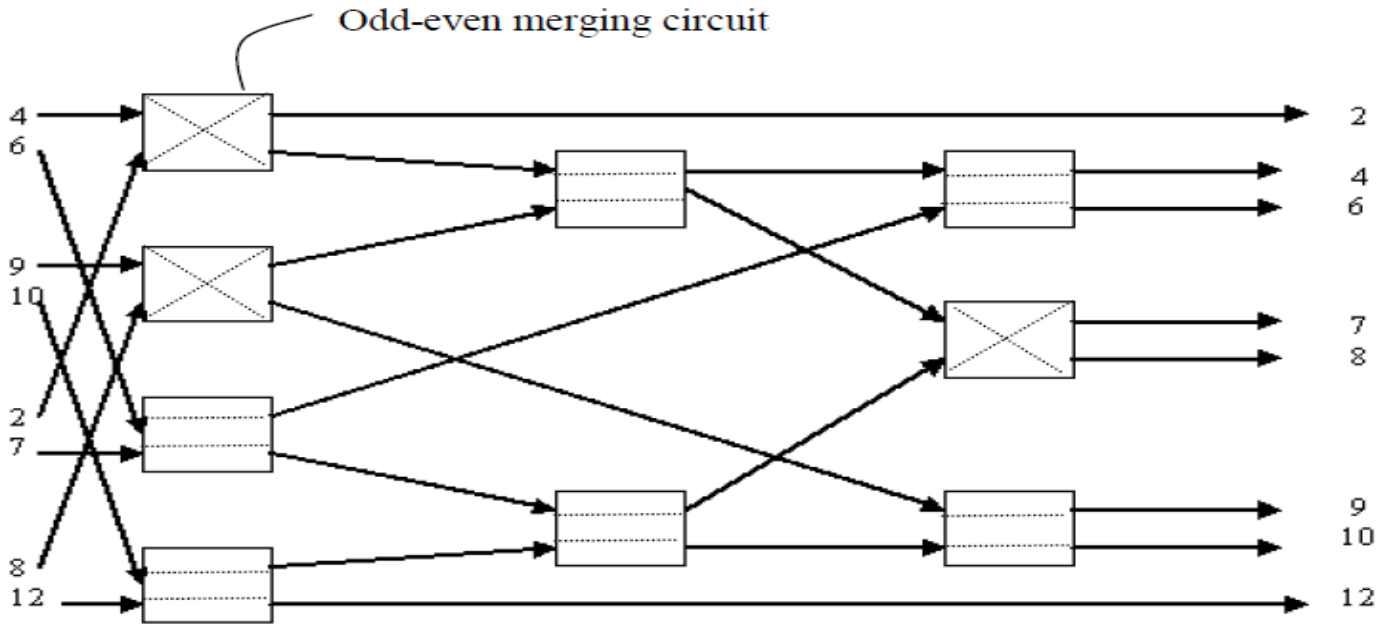


Figure 7: Merging Circuit

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 4th 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(log2 n).

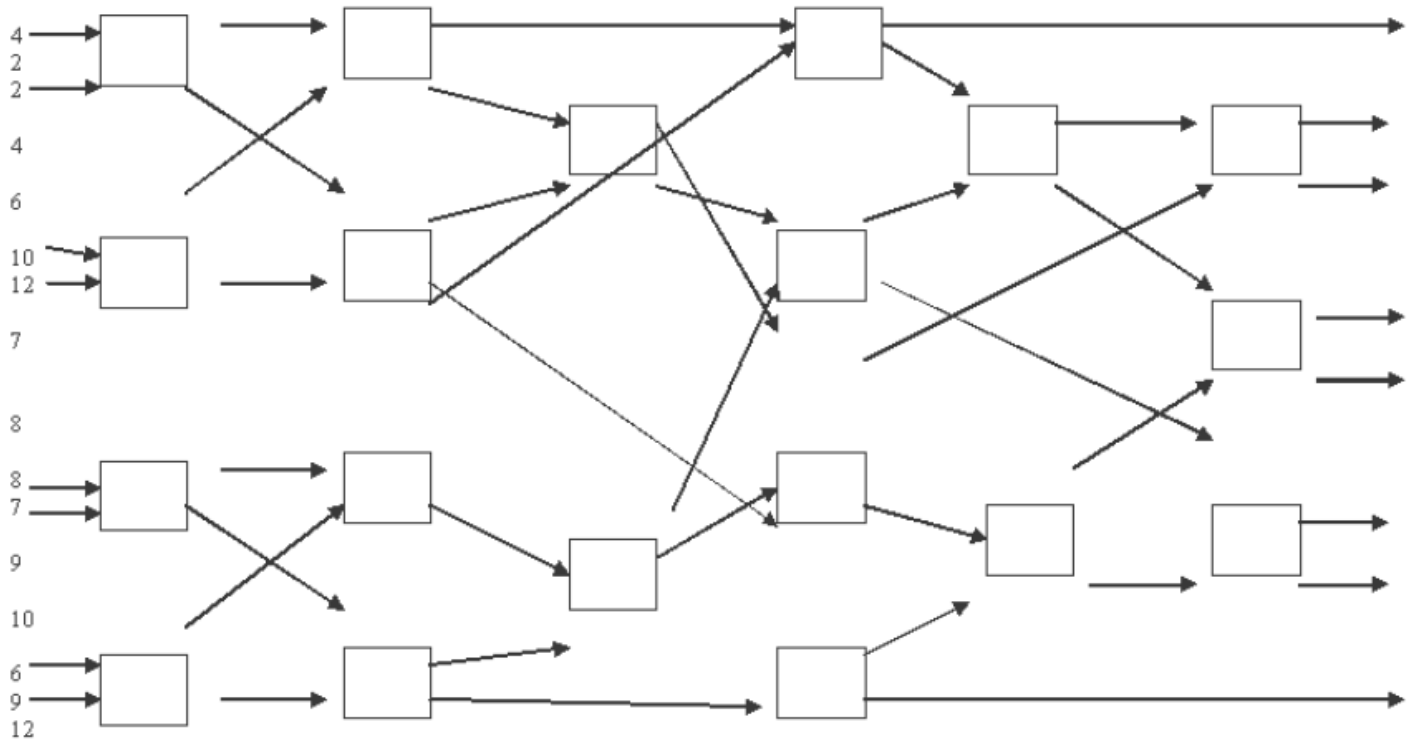


Figure 8: Sorting + Merging Circuit

b) Write short notes for any two of the following data structures for parallel algorithms
(i) Linked list
(ii) Array pointers
(iii) Hypercube

Ans:- (i) 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.

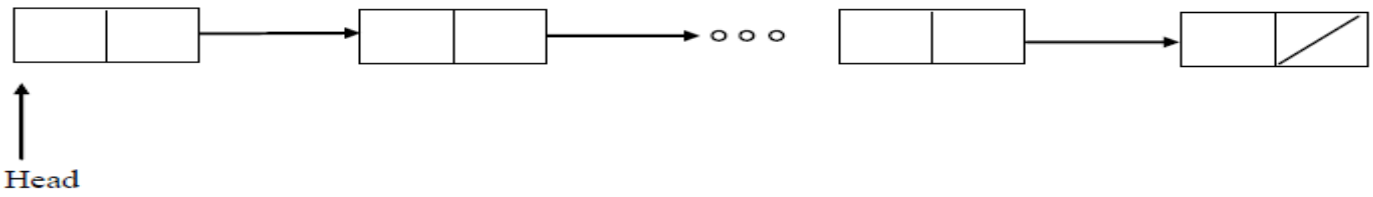


Figure 3: Linked List

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.

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:

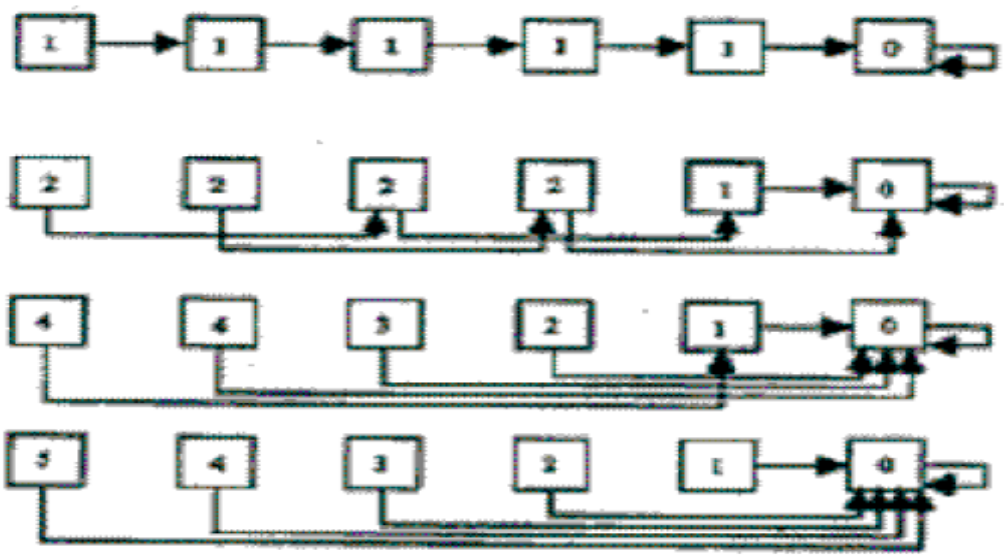


Figure 4 : Finding rank of elements

(ii) Array 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 **PRAM Algorithms** th element in

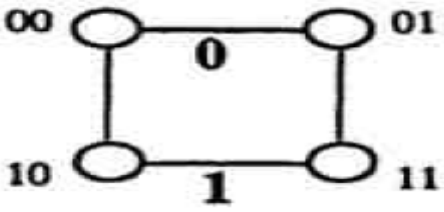
Fortran 90 is A(i) but in C it is A[i-1]. Arrays may be onedimensional or they may be multi-dimensional. General form of declaration of array in Fortran 90 is *type, DIMENSION(bound) [,attr] :: name*

for example the declaration

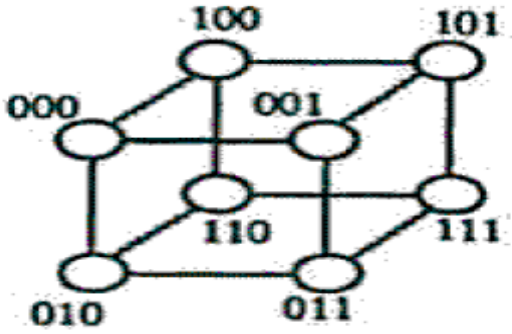
INTEGER, DIMENSION(5): A
declare an array A of size 5.
General form of declaration of array in C is
type array_name [size]
For example the declaration A
int A[10]
declares an array of size 10.

(iii) Hypercube : The hypercube architecture has played an important role in the development of parallelprocessing 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. Ancube can be defined recursively as depicted below:

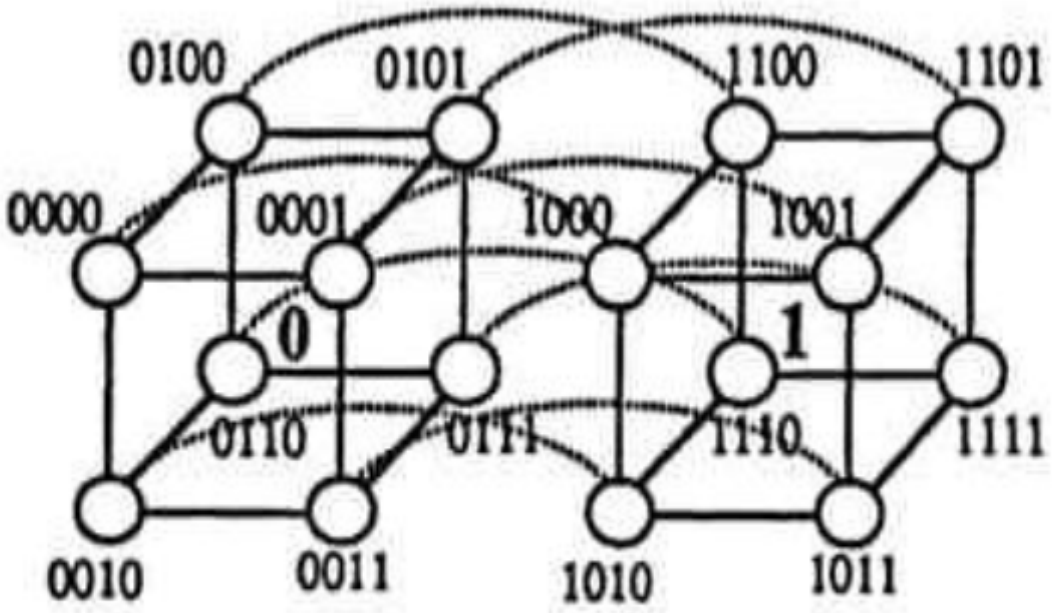
1-cube built of 2 0-cubes
Figure 5(a): 1-cube



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. **PRAM Algorithms**
- 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 too 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.

Q.7. a) Discuss synchronization principle for parallel for multi-processing environment.?

Ans:- 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.

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.

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.

b) In High-performance Fortran, write a FOR ALL statement to set Lower triangle of a matrix to zero.

Ans:- ***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.**

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.

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 Parallel Programming allocate that part of the computation to the processor holding the operands, whereupon no remote memory accesses will be involved.

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

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) Parallel Programming statement

where triplet has the general form subscript = lower: upper : step-size

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.

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)

c) Write a pscudo-code to find the product f(a) * f(B) of two functions in shared memory programming using library routines.

Ans:- 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**.**

Let us write a pseudocode to find the Product of the two functions f(A) * f(B). In the first algorithm we shall not use locking.

```
Process A                    Process B
prod = 0                      :
:                             :
fork B                        prod = prod* f(B)
:                                    :
prod = prod * f(A)           end B
:
join B
:
end A
```

If process A executes the statement prod = prod * 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 prod = prod* f(B) before process A could write result into the main memory. Then the Product contains only f(B) which is incorrect. To avoid such inconsistencies, we use locking.

```
Process A                    Process B
prod = 0                     :
:                            :
:                            lock prod
fork B                       prod = prod* f(B)
:                            unlock prod
lock prod                    :
prod = prod + f(A)           end B
unlock prod
:
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.

Q.8. *a)* Discuss the following recent parallel programming models:
(i) Threads model
(ii) Data Parallel model
(iii) Single Program Multiple Data

Ans:- (i) 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.

(ii) 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.

(iii) Single Program Multiple Data : 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.

***b)* Discuss briefly factors affecting parallel overheads**

Ans: parallel overheads : 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.

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 Evaluations
2) Cost involved in inter-processor communication
3) Synchronization
4) Parallel Balance Point

1) Uneven load distribution Evaluations: 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) 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.

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.

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.

***c)* Discuss various kinds of metrics involved for analysing the performance of parallel algorithms for parallel computers.**

Ans:- Analysing the performance 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:

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

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 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.

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,

$$O(g(n)) = \{ 0 <= c*g(n) <= f(n) \text{ for all } n >= n_o \}.$$

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.

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

2 The Total Number of Processors Required: 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).

3 The Cost Involved : 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.

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