UNIT II

CHAPTER 2

Parallel Programming

Syllabus Topics

Principles of Parallel Algorithm Design : Preliminaries, Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing, Methods for Containing Interaction Overheads, Parallel Algorithm Models, The Age of Parallel Processing, the Rise of GPU Computing, A Brief History of GPUs, Early GPU.

Syllabus Topic : Principles of Parallel Algorithm Design - Preliminaries

2.1 Principles of Parallel Algorithm Design

Q. 2.1.1 Explain Principles of Parallel Algorithm Design. (Refer sections 2.1, 2.1.1 and 2.1.2) (5 Marks)

- An algorithm provides step by step solution of the given problem. An algorithm accepts inputs from the user and based upon the input after performing the defined computations provides the output.
- In the similar fashion particularly a parallel algorithm accepts inputs from the user and execute several instructions simultaneously on different processing units and all separate outputs produced from different units are combined to provide the overall output of the algorithm. In this chapter we will discuss several techniques required to deal with the design of parallel algorithms.

2.1.1 Preliminaries

- The basic steps in the design of parallel algorithm are :
  (i) Partitioning of overall computation into smaller computations and
  (ii) Assignments of these smaller computations into different processors.
- These two key terminologies are explored in this section and some of the basic terminologies associated with the parallel algorithm design are also focused at significant levels.

2.1.2 Decomposition, Tasks, and Dependency Graphs

Q. 2.1.2 Explain Decomposition, Tasks, and Dependency Graphs. (Refer section 2.1.2) (7 Marks)

1. Decomposition

- The computer system is used to solve a problem by performing some computations based on input data and provides output data for further activities.
- The overall computation can be partitioned into number of small size computations so that these computations execute in parallel.
- The decomposition deals with the approaches of partitioning the overall computation into sub parts.
- When a computation is divided into many small tasks, it is referred as **fine - grained decomposition**. On the other hand the **course - grained decomposition** contains a small number of large tasks.

2. Tasks

- As we know that the operating system provides an environment for program developments and executions.
- In fact it is considered as an overall controller of the program. In programming the basic unit of computation is referred as a task and is controlled by an operating system.
- In the context of parallel programming the tasks are units of computation based upon that the overall computations is decomposed.
- The problem to be solved using the parallel programming approach is divided into arbitrary sized multiple tasks for simultaneous executions so that the computation time is minimized effectively.

3. Task - dependency graph

- Is a directed acyclic graph. Typically a graph is a collection of nodes and edges, the task - dependency graph also contains nodes and edges.
- The nodes in the task - dependency graph represent tasks whereas edges between any two nodes represent dependency between them. For example, there is an edge exists between two nodes **T1** and **T2**, if **T2** must be executed after **T1**.

Ex. 2.1.1

Consider the task - dependency graph shown in Fig. P. 2.1.1 and find out the following :

(i) Maximum degree of concurrency

(ii) Critical path length

(iii) Total amount of work

(iv) Average degree of concurrency

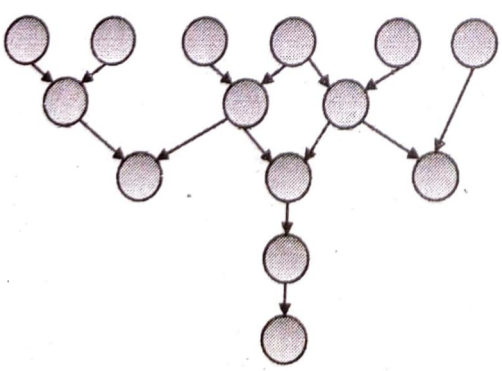


Fig. P. 2.1.1 : Task - dependency graph

Soln. :

(i) Maximum degree of concurrency

The maximum number of tasks allowed to execute in parallel is termed as the maximum degree of concurrency. In the given graph the maximum degree of concurrency is **6.**

(ii) Critical path length

- There are two types of nodes included in the task - dependency graph namely **start node** and **finish node**.
- Start nodes are nodes with no incoming edges whereas nodes with no outgoing edges are called as finish nodes.
- The **critical path** in a task - dependency graph is the longest directed path between any pair of start and finish nodes.
- The quantity referred as critical path length is the sum of the weights of the nodes on a critical path.
- In the given graph the critical path length is **5.**

(iii) Total amount of work

Here, the total amount of work is **14** if it is assumed that the each task takes one unit of time.

(iv) Average degree of concurrency

- The average number of tasks allowed to execute in parallel is called average degree of concurrency.
- It is a more useful measure and is computed using the following formula.

  (Average degree of concurrency = Total amount of work /Critical path length)
- In this example average degree of concurrency = 14/5 = **2.8**

☞ Example : Data Decomposition

- The matrices A and B are divided into number of rows and columns. The divisions of rows and columns are in continuous sequences and termed as strips.
- The matrices A and B are multiplied and the resultant matrix C is generated.
- The elements of the C matrix can be computed independently according to the definition of matrix multiplication.
- This fact suggests the possibilities for performing matrix multiplication in parallel.

- The parallel computations can be performed using dividing the overall computation into number of subtasks.
- In this case each subtask should contain a row of the matrix A and a column of the matrix B.
- The total number of subtasks included to perform the multiplication appears to be equal to N^2 according to the elements in resultant matrix C.
- The necessary computations of the basic subtasks can be performed when the required data sets are available.
- The required data sets are a row of the matrix A and all the columns of matrix B for basic subtasks.
- The simple solution in this case is duplicating the matrix B in all the considered subtasks for multiplications.
- This solution is unacceptable because of the extra memory requirements for extra data storage. The solution for this problem should have the data availability only required for the computations.
- The algorithm for the matrix multiplication with the solution of the problem mentioned here is an iterative procedure.
- In this procedure the number of iterations is equal to the number of subtasks. In this case at each iteration of the algorithm a row of matrix A and a column of matrix B are contained in each subtask.
- The subtasks containing rows and columns computes scalar products at each iteration and the corresponding elements of the matrix C are generated.
- After completing of all iteration computations the columns of matrix B must be transmitted so that subtasks should have new columns of the matrix B and new elements of the matrix C could be calculated.
- This transmission of columns among the subtasks must be executed in such a way that all the columns of matrix B should have appeared in each subtask sequentially.
- Fig. 2.1.1 shows the iterations of the matrix multiplication algorithm where four rows and four columns are assumed.
- In this scenario at the beginning of the computations each subtask contains i^{th} row of A matrix and i^{th} column of B.
- As a result the subtask *i* can compute the element C_{ii} of the result matrix *C*. Further each subtask transmits its column of matrix *B* to the following subtask in accordance with the ring structure. These actions should be repeated until all the iterations of the parallel algorithm are completed.

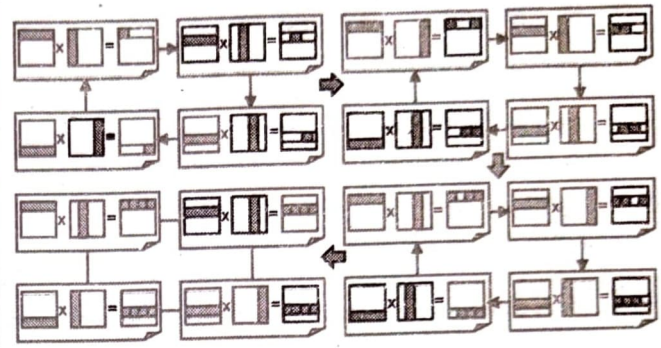


Fig. 2.1.1 : General scheme of data communications for the parallel matrix multiplication algorithm using block striped decomposition

2.1.3 Granularity, Concurrency and Task-Interaction

2.1.3(A) Granularity

- Granularity is the descriptions about a system in terms of how divisible it is. There are two types of granularity namely **fine - grained** and **course - grained**.
- Fine - grained system has a high - granularity. This means in fine - grained an object (or system) is divided into larger numbers of smaller parts.
- In the similar ways a course grained refers to the system divided into a smaller number of larger parts.
- For example, the use of grams to represent weight of an object is more granular than using kilograms to represent the weight of the same object.
- In general parallelism is categorized with instruction and data stream in parallel computing. Granularity in parallel computing is an additional way of categorizing parallel computation.
- Term granularity is used in parallel computing to refer about the division of overall task into number of subtasks.
- The fine - grained in parallel computing refers the division of a task into a large number of smaller subtasks.

- In the similar fashion course - grained describes the division of a task into larger number of longer subtasks. In parallel computing, technically a measure of the computational work is called as a grain. In fact a grain represents the ratio of computation work to communication work.

☞ **How to measure granularity ?**

- There are three relative values actually used to specify the granularity : **fine, medium, coarse**. In the parallel algorithm the granularity is generally used to describe about a parallel section of algorithm.
- There are three characteristics of the algorithm used for the determination of granularity. It includes the hardware used for the running of the algorithm.
- The first characteristic is **the structure of the problem**. The data parallel programming works in the form where specified operations are performed on many pieces of data.
- These operations are executed by different processing elements and communications amongst the PEs required.
- In association with granularity the task in data parallel computation is considered to have small granularity or **fine - grained**.
- On other cases like the executions of large subroutines independently with little communications with each other is considered as **coarse - grained**.
- The second characteristic is the size of the problem. This can be described using an example where 10 numbers are to be incremented. The number of processing elements in this case is assumed as 10 then the algorithm requires 1 clock cycle for computation.
- Later on assume the increase in the problem size and therefore 100 numbers are to be incremented. This way the task size has been increased and this is considered as a coarser granularity.
- The third characteristic to deal with the granularity in the parallel algorithm is the number of processors available. The number of processors is reduced for the problem size the task size increases and the granularity becomes coarser.

2.1.3(B) Concurrency

- The tendency for the events in real world to happen at the same time is called as concurrency.
- The concurrency is one of the natural phenomena in the real world because at a particular instance of time many things are happening simultaneously. The concurrency is required to deal with for designing the software system for real world problems.
- There are generally two important aspects when dealing with concurrency for real world problems : ability of dealing and responding external events occur in random order, and required to respond these events in some minimum required interval.
- It is really simple to handle when different activities happen in a truly parallel fashion by creating separate programs to deal with each activity.
- The real challenge occurs when required to design concurrent system in which interactions among the concurrent activities need to be co-ordinated.
- The concept termed as degree of concurrency is related to granularity is used in the algorithm design.
- The maximum degree of concurrency is the maximum number of concurrent tasks can be executed simultaneously in a program at any given instance of time.
- Another term related to concurrency is the average degree of concurrency which is used to refer the average number of tasks can be processed in parallel during the execution of the program.

2.1.3(C) Task-Interaction

- The parallel executions of tasks provide efficiency in terms of speedup because tasks in a parallel algorithm simultaneously execute in different processors.
- There are basically three factors granularity, concurrency and interaction are responsible to affects the speedup of the parallelization.
- The interaction provides the communication among the tasks running in different processors.
- The interactions among the tasks included for providing the solution of a problem is required because the tasks share input, output, or intermediate data.
- The task - dependency graph shows the dependencies because output of one task can be the input of another task. This means the second task cannot proceed unless the output of the first task becomes available.

2.1.4 Processes and Mapping

2.1.4(A) Processes

- A problem to be solved using the parallel programming approach is considered to be divided into number of parts. Each part can be performed independently and is called as a task.
- In the context of this discussion a process is an entity responsible for performing the assigned task.
- This process is an abstract idea about doing the computations based upon the steps included in a task and input data available for it for producing the output.
- The environment in which a problem is being solved by means of parallel computations usually has several running process simultaneously.
- These processes communicate with each other for the need of synchronization and exchanging of information.
- The programs written for such platforms are called as parallel programs and one parallel program can have several processes for handling different tasks simultaneously.
- The Fig. 2.1.2 describes a problem divided into several tasks and assigned to separate processors for simultaneous executions.

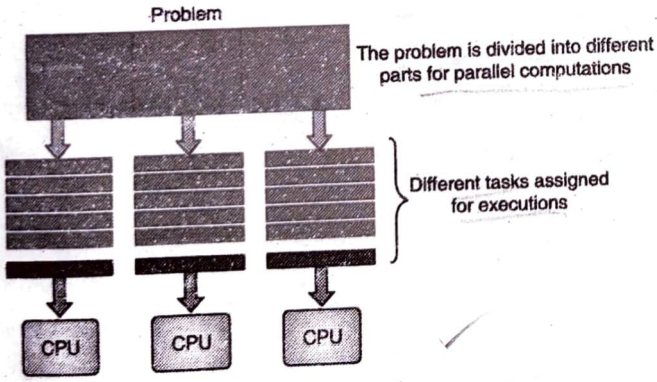


Fig. 2.1.2 : Parallel execution of tasks

2.1.4(B) Mapping

- In parallel program design it is required to specify where each task is to execute.
- As we have discussed in the previous section that a process performs a task. In this context the approach used for assignment of tasks to processes is called mapping.
- In fact mapping deals with two goals and these are referred as maximization of processor utilization and minimization of communication costs amongst processes.

2.1.5 Processes Versus Processors

- A program in execution is called as a process and in the design of parallel algorithms it is considered as the entity responsible for performing tasks.
- The computations defined to be performed in the tasks are executed by the physical unit called as processor. In this chapter the processes are used to describe parallel algorithms and programs.
- In the field of parallel computing it is assumed that the number of processes depends upon the total number of processors exist and most of the time it is considered as each process executes in a separate processor.
- In the parallel algorithm design if the algorithm is very complex and requires to have many processes so it is assumed to have running in the form of pseudo parallel processing so that the availability of processors should not resist in design.

Syllabus Topic : Decomposition Techniques

2.2 Decomposition Techniques

Q. 2.2.1 Write short note on Data-decomposition and Recursive decomposition. (Refer sections 2.2.1 and 2.2.2) (7 Marks)

- The problem required much more time for computations are possible to divide multiple parts and each and every part is the implementation of a particular task.
- The task - dependency graph used to define a set of tasks and these tasks are the candidate for concurrent executions.
- There are various techniques commonly used for decomposition and in this section some of the techniques are covered.

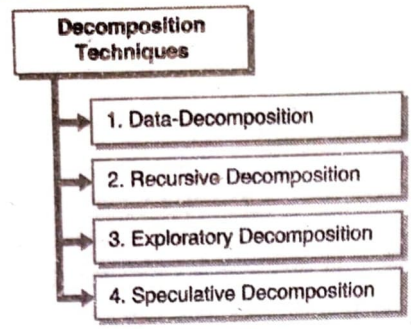


Fig. 2.2.1 : Decomposition techniques

2.2.1 Data-Decomposition

- The large data sets involved in the problems can be divided into smaller parts and processed independently by several computers concurrently.
- The computations of partitioned data on computers are usually used for some kinds of analysis.
- The problem is classified as an embarrassingly parallel if the subset of the data can be analyzed independent of the rest of the data. The data - decomposition is a common technique used for concurrent processing of the data.
- There are basically two steps used in this technique.
- In the first step the overall input data required for performing the defined computation is divided into multiple parts.
- The second step is based upon the division of overall computation into number of tasks handled on the portioned data.
- The actual operations implemented on these tasks are similar but the data upon which these operations work are different.

Example : Matrix multiplication

Consider the problem of multiplying two n × n matrices A and B to yield matrix *C*. The output matrix *C* can be partitioned into four tasks as given below. Here each task computes one element of result matrix.

$$\text{Matrix A} \rightarrow \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \quad \text{Matrix B} \rightarrow \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}$$

$$\text{Matrix C} \rightarrow \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \times \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}$$

$$\text{Matrix C} \rightarrow \begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix}$$

$$\begin{bmatrix} a11.b11 + a12.b21 & a11.b12 + a12.b22 \\ a21.b11 + a22.b21 & a21.b12 + a22.b22 \end{bmatrix}$$

2.2.2 Recursive Decomposition

- As we know that the Divide and conquer is one of the strategy of solving the computational problems.
- The divide and conquer strategy is based on two ideas : first approach says to solve the problem directly if it is trivial and in the second approach, decompose the problem into smaller parts if it cannot be solved as it is and solve the smaller parts.
- The recursive decomposition is based on providing concurrency in problems that can be handled in divide and conquer strategy.
- The problem to be solved using recursion is divided into multiple sub problems provided that each of these sub problems is independent.
- Each sub problem can be solved individually by dividing further into other sub problems and solved using recursion.
- In programming a function or procedure calls it-self is called as recursion. For example in the following program the function demo is a recursive procedure.

```
main()
{
        int i;
        i = 10;
        demo(i);
}
demo(int count)
{
        count--;
        printf("The value of the count is %d\n",count);
        if (count > 0)
        demo(count);
        printf("The count is %d\n",count);
}
```

Example : Merge sort

- The example of merge sort as shown below is implemented in sequential and parallel version thereafter. In this example the array is first divided into two parts then sorted these two parts recursively. The sorted parts are finally merges to produce the final output. The overall computations are organized here in the binary tree.

- The parent process provides array to each process for sorting of elements. The process divides the array into two halves and sends to the children. The children perform their part of computations and send the sorted array back to the parent. The merging of these elements is performed by the parent and sends the array back up in the tree.

☞ **Merge Sort : Pseudo code for sorting of an array using merge sort sequentially**

```
void mergeSort(int* a, int first, int last, int* aux)
{
    if (last <= first)
    return;
    int mid = (first+last)/2;
    mergeSort(a, first, mid, aux);
    mergeSort(a, mid+1, last, aux);
    mergeArrays(a, first, mid, a, mid+1, last, aux, first,
    last);
    for (int i=first; i<=last; i++)
    a[i] = aux[i];
}
void mergeArrays(int* a, int afirst, int alast, int* b, int bfirst,
int blast, int* c, int cfirst, int clast)
{
    int i=afirst, j=bfirst, k=cfirst;
    while (i<=alast&& j<=blast)
{
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
        while (i<=alast)
            c[k++] = a[i++];
        while (j<=blast)
            c[k++] = b[j++];
}
```

☞ **Merge Sort : Pseudo code for sorting of an array using merge sort in parallel**

```
void parallel_mergeSort()
{
    if (proc_id> 0)
    {
        Recv(size, parent);
        Recv(a, size, parent);
    }
    mid = size/2;
    if (both children)
    {
        Send(mid, child1);
        Send(size-mid, child2);
        Send(a, mid, child1);
        Send(a+mid, size-mid, child2);
        Recv(a, mid, child1);
        Recv(a+mid, size-mid, child2);
        mergeArrays(a, 0, mid, a, mid+1, size, aux, 0, size);
        // declare aux local
        for (int i=first; i<=last; i++)
        a[i] = aux[i];
    }
    else
        mergeSort(a, 0, size);
    if (proc_id> 0)
        Send(a, size, parent);
}
```

2.2.3 Exploratory Decomposition

Searching smaller parts concurrently

- There are situations where the problem decomposition goes hand in hand with its executions.
- Such problems typically involve the exploration or search of a state space of solutions. The search space of the problem is divided into smaller parts and each smaller part is searched concurrently till the point at which the expected solution is found.

☞ **Example of exploratory decomposition**

Consider a state space searching problem such as finding a solution to a puzzle problem. The steps of solving such problems using the exploratory decomposition are as follows :

- The computations required for the decomposition can be divided into multiple tasks where each task is searching for a different portion of the search space.

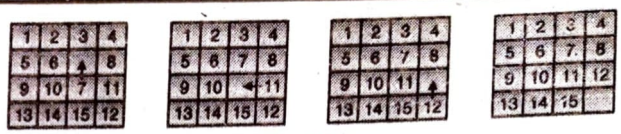


Fig. 2.2.2

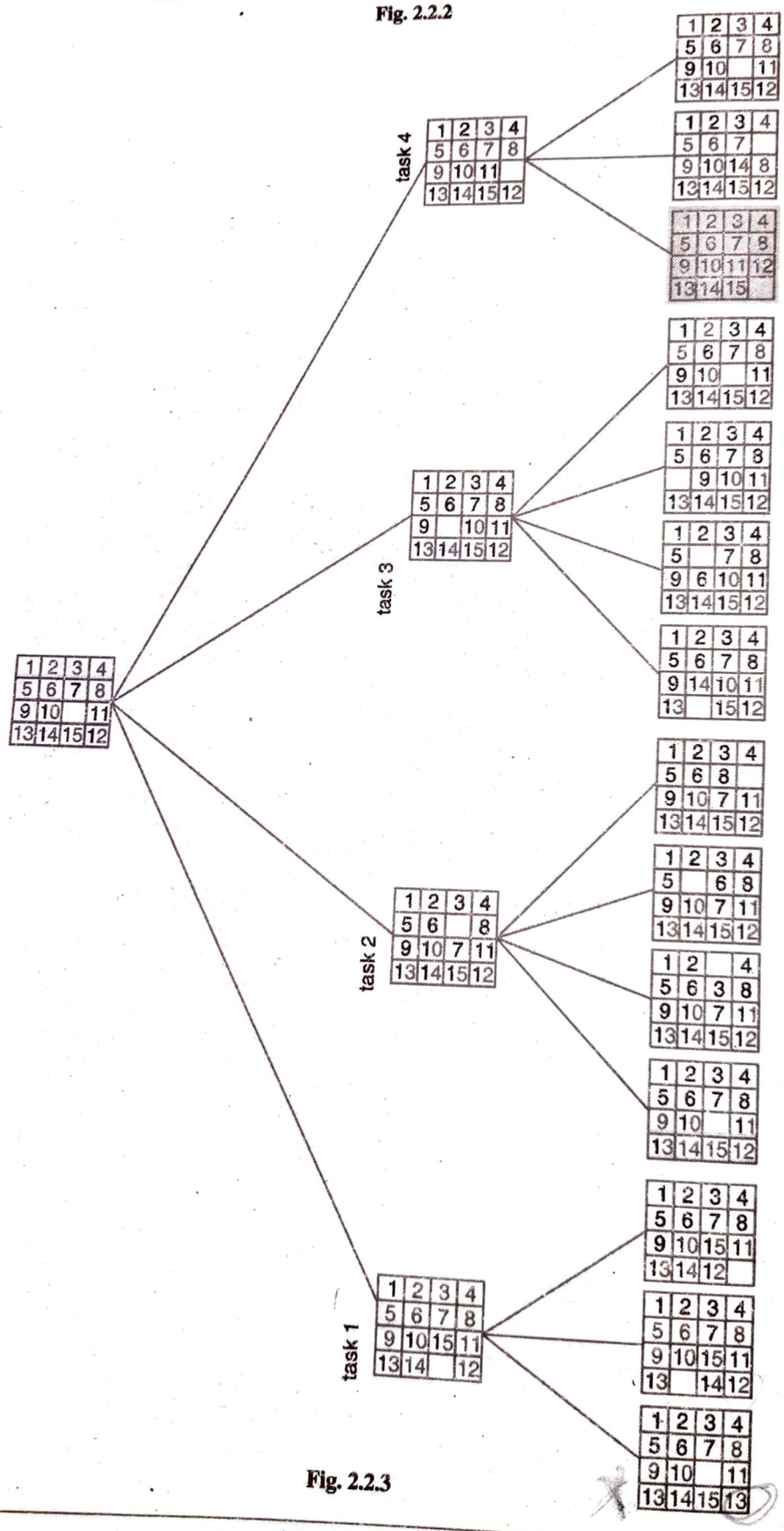


Fig. 2.2.3

- The task of finding the shortest path from initial to final configuration now translates to finding a path from one of these newly generated nodes to final configuration.
- The 15 puzzle problem is typically solved using tree search techniques. Starting from initial configuration, all possible successors are generated.
- In the 15 puzzle problem there are 15 tiles numbered 1 through 15 are arranged in a 4 × 4 grid as shown in Fig. 2.2.2. One tile is left blank so that moves can be made.
- The four possible moves here are represented as moves up, down, left and right. The initial as well as the final configurations are specified.
- This problem is characterized by the objective of determining any sequence of moves or a shortest sequence of moves.
- The solution of the problem must be searched from an arbitrary state.
- The solution of the 15 puzzle problem is provided using the tree - search techniques. Once we start from the initial configuration all the configurations possible as successor configurations are generated.
- This is handled using two possibilities: one says about to occupy empty slot by any one of the neighbour present, and second is to find out a path from one of the new configurations to the final one.

A state space graph

- Fig. 2.2.3 shows the configuration space generated by the tree search. As we know that any graph is a collection of nodes and edges the state space graph also contains nodes and edges.
- In this graph the nodes are used to represent configurations and edges represent connections of configurations. Here every edge of the graph connects configurations that can be reached from one another by a single move of a tie.

2.2.4 Speculative Decomposition

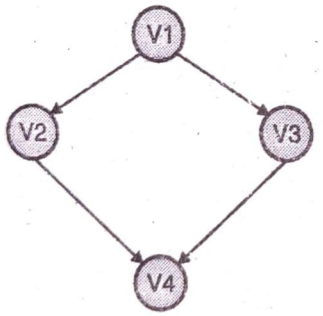
- The speculation decomposition technique is used in a situation when a program has many options to take in terms of branches based upon the outputs of other parts that preceded it.
- The situation can be described in terms of specific task, consider a task T1 involves in doing the computation C1 and going to produce the output O1.
- The next computation to be performed may be decided based upon the output of task T1. In the similar fashion other tasks can start executions concurrently in the next stage.
- The situation can be better compared with the switch statement in C programming.
- The switch statement works based upon the value of the expression on which it is based and only the corresponding case statement executes. Some or all of the cases execute in advance in speculative decomposition.
- The situation in which value of the expression is known then the results from the computation to be executed in that case will be kept.
- The anticipation about the possible computations causes the performance gain in the statement executions. For example consider the following switch statement in sequential and parallel versions.

Sequential	Parallel version

compute expr;
 	
switch (expr)
{
case 1 : compute-e1;
break;
case 2 : compute-e2;
break;
case 3 : compute-e3;
break;
.....
}
	Slave(i)
	{
	compute ei;
	Wait(request);
	if (request)
	Send(ei,0);
	}
	Master()
	{
	compute expr;
	switch (expr)
	{
	case i :
	Send (request, 1);
	Receive(a1,i);
	....
	}
	}

Example : Topological sorting

- In a directed acyclic graph topological sorting is used to provide an ordering of the vertices. For example consider two vertices U and V of a graph G.
- In this graph when a path exists between U and V, then V appears after u in the ordering.
- It is a requirement that the graph should have acyclic property, otherwise for an edge represented as (U, V) there would be a path from U to V and also from V to U, and therefore the ordering cannot be obtained.
- Now let's assume there are a number of tasks we need to be performed in which some of the tasks depend on the others and it is possible to do only one at one.
- These tasks can be organized in the dependency graph. Here one must be able to find out an ordering of the tasks based on the dependencies. Consider Fig. 2.2.4 for understanding of topological sorting.



The legal orderings are → V1, V2, V3, V4
and V1, V3, V2, V4

Fig. 2.2.4

Steps of the topological sorting algorithm

Step I :	Initially compute the in – degrees of all the vertices in the graph.
Step II :	Find U as a vertex with degree 0 and store it in the list for ordering.
Step III :	At this stage if no such vertex is detected then there is a cycle found and the algorithm stops
Step IV :	Remove the vertex U and all the edges (U, V) it belongs from the graph
Step V :	At this step need to update in – degrees of the vertices remains
Step VI :	Repeat step 2 through 4 till the vertices present for processing

- In the topological sorting algorithm we can get the idea about the next vertex in the order before the next trace for the vertex is actually performed.

Example based on Topological sorting

Ex. 2.2.1

Now we will discuss another example to understand topological sorting. Consider the given directed graph $G = (V, E)$, find a linear ordering of vertices such that : for all edges (v, w) in E, v precedes w in the ordering.

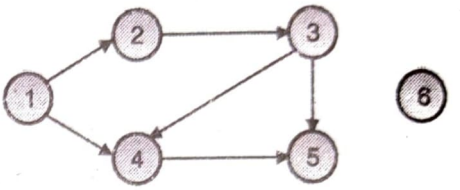


Fig. P. 2.2.1

Soln. :

For the topological sorting any linear ordering in which all the arrows go to the right is a valid solution.

This way the linear ordering shown in Fig. P. 2.2.1(a) is a valid solution.

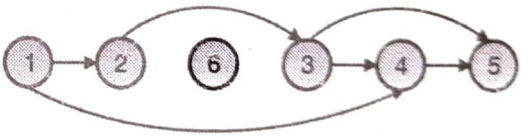


Fig. P. 2.2.1(a) : Valid topological sorted graph

The graph shown in Fig. P. 2.2.1(b) is not a valid topological sorted order.

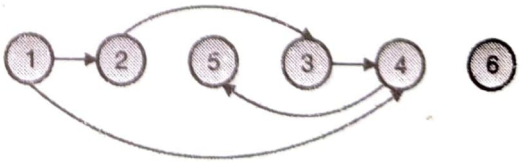


Fig. P. 2.2.1(b) : Invalid topological sorted graph

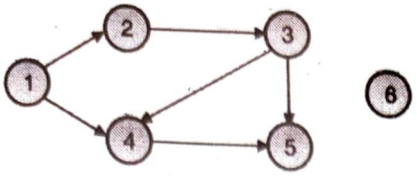
Topological sort algorithm : The steps of the topological sorting algorithm are given below.

Step 1 : Identify vertices that have no incoming edges (select one vertex)

Step 2 : Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.

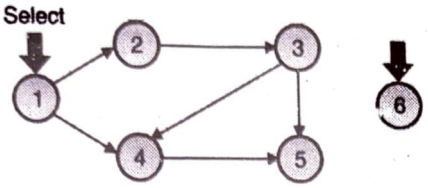
Step 3 : Repeat Steps 1 and Step 2 until graph is empty.

Now the steps of the algorithm are performed on the given graph.

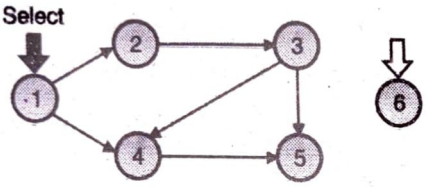


Perform Step 1 : Identify vertices that have no incoming edges.

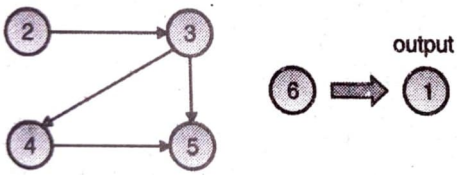
This means select vertices with in–degree zero.



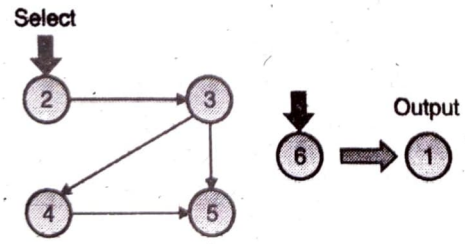
Perform Step 1 : Identify vertices that have no incoming edges. (Select one vertex)



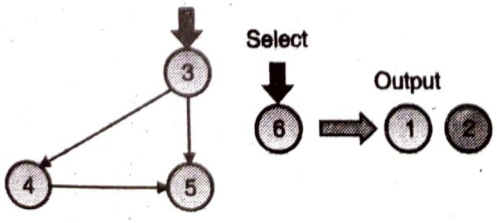
Perform Step 2 : Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



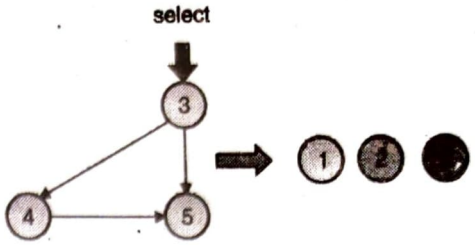
Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty

Final Result

Finally the result of the topological sorting is shown here.

Syllabus Topic : Characteristics of Tasks and Interactions

2.3 Characteristics of Tasks and Interactions

- The different decomposition techniques are used to divide the overall problem into several concurrent tasks so that they can be executed in parallel.
- The further step of the parallel algorithm design deals with the mapping of these tasks into available processes.
- The decomposition approach provides the possibilities of writing good parallel programs to be performed in efficient time.
- The communications amongst tasks play very crucial roles while mapping the tasks into processes for parallel executions.
- This part describes properties of tasks and inter-task communications among them for good mapping for better performance oriented parallel algorithm designs.

2.3.1 Tasks Characteristics

Q. 2.3.1 What are the Characteristics of Tasks? (Refer section 2.3.1) (5 Marks)

- There are some key characteristics used to influence the choice of mapping of tasks onto available processes and at the same time the performance of the parallel algorithm is also maintained.

- The four basic characteristics are Generation of task, Size of tasks, Knowledge of Task Sizes and Data size associated with Tasks.

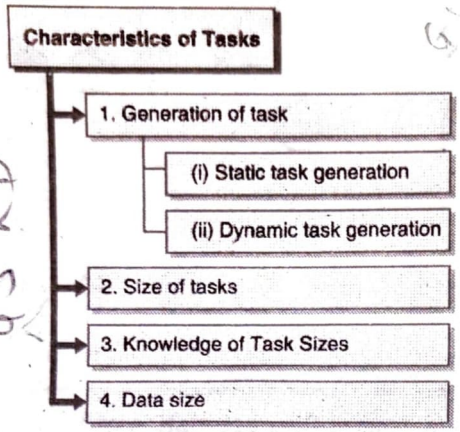


Fig. 2.3.1 : Characteristics of Tasks

➜ 1. Generation of task

As we know an algorithm consists of tasks and in a parallel algorithm these tasks are considered to be executed in parallel. The tasks included in a parallel algorithm are possible to generate either on a prior basis termed as static or created whenever required termed as dynamically.

➜ (i) Static task generation

- The approach in which the work(s) to be performed by an algorithm is known in advance on priory basis is called as static task generation.
- The tasks are defined before starting the execution of the algorithm and the specified order is to be followed by the algorithm in execution.
- For example consider the matrix multiplication algorithm where overall matrices divided into different chunks and distributed to various processors in a parallel machine. In each processor, the task for matrix multiplication executes on different sets of data considered for operations.

➜ (ii) Dynamic task generation

- The tasks to be performed are created dynamically based upon the decomposition of the data in certain situations. In these cases the tasks to be performed are not available before algorithm executions.
- The recursive and exploratory decomposition techniques are considered as examples of dynamic task generation.

➜ 2. Size of tasks

- Every task takes some amount of time for its completion. The size of a task is represented by the time required for its completion.
- There are two different categories in which programming tasks can be divided : ***uniform*** and ***non-uniform***.
- The tasks are required to map into available processes so that it can be computed and it depends upon the type of the tasks.
- There are different mapping schemes and uniform tasks takes almost same amounts of time in mapping. In the case of non uniform tasks, the mapping times required varies in different schemes.

➜ 3. Knowledge of Task Sizes

- The task size knowledge is used for the choice of mapping scheme. The tasks are mapped into processes and the prior knowledge of the tasks used in mapping.
- As an example the knowledge of the computation time required for each task is used to apply different decomposition techniques.
- In the similar fashion in some of the examples where dynamic decision are required during processing the prior knowledge of the task size is not known.

➜ 4. Data size

- The data size is one more crucial property associated with the task. The required data for performing a task should be made available when mapping it into processes.
- The overheads associated with the data movement can be reduced when the size of the data as well as its memory location are available.

2.3.2 Characteristics of Inter-Task Interactions

- As we know a parallel algorithm contains many tasks and these tasks may have communications with each other to share some information.
- The exchange of information amongst tasks is done by the use of mechanism called as inter - tasks communications.
- The interactions amongst the tasks depend upon the types of interactions in different algorithms.

- The different programming paradigms and mapping schemes may support different natures of interaction schemes.
- This indicates that some of the interactions are suited to some techniques and other works most efficient with other techniques.

2.3.3 Static Versus Dynamic Pattern

There are two classes of interactions amongst the concurrent tasks based upon the types of pattern used in interactions. These patterns are static and dynamic patterns.

1. Static interaction pattern

- The static pattern specifies that the times at which interaction among tasks occur is predetermined. The set of tasks to have interactions with each other is also known before the execution of algorithms in static pattern.
- The message - passing paradigm like MPI(Message Passing Interface) and PVM (Parallel Virtual Machine) can be easily used to create applications with static interaction of tasks.
- The message - passing approach provides the specifications for having association of sender and receiver tasks in message exchange.

2. Dynamic interaction pattern

- The interaction times of the tasks to be interacted as well as a set of tasks to have interactions decided at the algorithm execution time only in dynamic interaction pattern.
- It is always a daunting task to program dynamic interaction of tasks because of unpredictable nature of dynamic communications.
- The synchronization between a sender and receiver is one of the challenging issues. This may create problem if not handled properly particularly in a situation where both sender and receiver communicate at the same time.

2.3.4 Regular Versus Irregular

- The interactions are also placed in different classes based upon their spatial structure.
- The interactions are kept in regular and irregular classes in consideration with spatial structures.
- The interaction pattern is considered as **regular** if it is possible to exploit some of the structure of the interaction pattern for efficient implementation.
- In the case where no such pattern exist called as **irregular** pattern. Like dynamic pattern of interactions the irregular pattern is also difficult to implement particularly in message - passing environment.

2.3.5 Read-Only Versus Read-Write

- Inter-task communication commonly called as task-interaction is used for data sharing among tasks. The choice of mapping of tasks into processes is affected by the type of sharing data between tasks.
- There are two different categories of data sharing can be considered in task - interactions : **read-only** and **read-write**.

1. Read - only interactions

- This is used in a situation where the data to be shared among concurrent tasks is required for read only purpose.
- The interactions among tasks in such situations are read - only interactions.
- Consider an array **A** with 10000 elements and these elements are added in parallel. In this example **A** is divided into multiple parts and the elements in each part are added concurrently.
- Here the input array is required for reading purpose only. In this example the tasks require to read array **A** for input purpose only.

2. Read - write interactions

This is required in a situation where many tasks have to perform read and write operations on some shared data.

2.3.6 One-way Versus Two-way

- The communications among tasks can also be categorized in terms of one way and two way communications. Suppose there is a pair of task in which one is the producer and another is the consumer.
- The interaction between producer and consumer task is termed as two - way because the data required by the consumer task is supplied by the producer task.

- The interaction between two tasks is typically called as **two - way interaction** when data required by one task is supplied by another task explicitly.
- The interaction is termed as **one - way interaction** when one task in a pair of tasks starts the communication and completes the interaction without interrupting other task.
- The read - only interactions in any form is considered as one - way whereas all read - write interactions can be considered as one - way or two-way depending upon the context.

Syllabus Topic : Mapping Techniques for Load Balancing

2.4 Mapping Techniques for Load Balancing

- The mapping techniques are used to map tasks into processes so that parallel executions can be performed.
- The overall computation associated in a problem to be solved divided into number of tasks. These tasks are mapped onto processes with the goal of completing executions in the minimum amount of time.
- This goal can only be achieved when the overheads associated with the parallel executions of tasks are minimized.
- There are two factors involved with the overheads in the parallel executions of tasks. The first factor due to which overhead occurs is the total time required in inter - process interactions during communications.
- One more factor responsible for overhead is the amount of time some of the processes wait without doing any significant works typically called as idleness. These factors provide the objectives for considering mapping as a good mapping.
- The mapping is assumed as a good mapping when the computations and interactions are well balanced at each stage of the parallel algorithm.

2.4.1 Mapping Techniques

Q. 2.4.1 What is Mapping Techniques? (Refer section 2.4.1) (4 Marks)

There are two different categories of mapping techniques are used in parallel algorithms : ***static*** and ***dynamic***. The suitability of these two mapping techniques is decided on the basis of tasks characteristics and how they interact with each other.

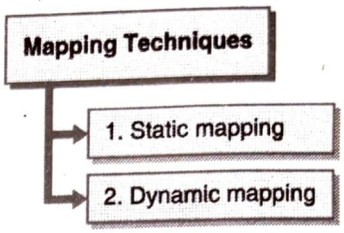


Fig. 2.4.1 : Mapping Techniques

➜ **1. Static mapping**

- In this technique, the mapping of tasks onto processes is performed before execution of algorithms. This indicates the tasks are distributed among available processes prior to execution of algorithms.
- The factors like knowledge of the task size, data size and inter - task interaction characteristics are used in deciding about the mapping technique to be used.
- For example, these two techniques static and dynamic mapping can be used when tasks are generated statically but which mapping will perform better is decided on the basis of task factors and parallel programming paradigm.

➜ **2. Dynamic mapping**

- In this technique, the mapping of tasks onto processes is performed during the execution of algorithms.
- This indicates the tasks are distributed among available processes when algorithm actually executes.
- There are basic situations where dynamic mapping is applied. The first and straightforward case is if tasks are generated dynamically then this mapping technique is used.
- Dynamic technique is applied in a situation where the large amount of data is associated with tasks. In this situation dynamic mapping moves data among available processes.

2.4.2 Schemes for Static Mapping

- The static mapping is usually used along with some partitioning techniques naturally. There are three different schemes suggested for static mapping :
- ***Mapping Based on Data Partitioning, Task Graph Partitioning*** and ***Hybrid Strategies.***

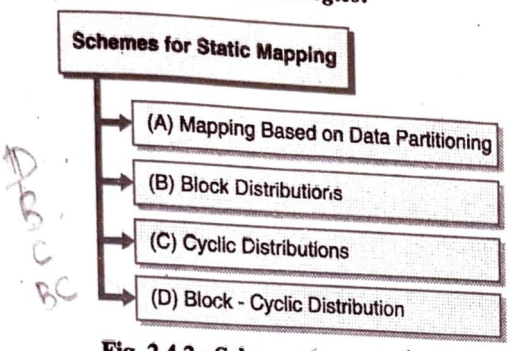


Fig. 2.4.2 : Schemes for static mapping

2.4.2(A) Mapping Based on Data Partitioning

- The mapping is used for associating tasks into processes. One of the popular rules used in High - Performance Computing called as owner computes rule and according to this rule mapping of tasks onto processes is similar to mapping of data to processes.
- The data associated with the algorithms are commonly represented by arrays and graphs. In this section we will discuss mapping of tasks onto processes on the basis of arrays and graphs.

Array Distribution Schemes

- The purpose of this scheme is to distribute array elements across local memories of a parallel computer.
- The direct benefit of this distribution is that the elements can be accessed in parallel for processing.
- There are three standard distributions of the dense arrays : **block, cyclic** and **block cyclic.**
- The rule commonly used and implemented in High Performance Fortran is named as Owner Computes rule.
- The owner computes rule is applied in a decomposition techniques based on the partitioning of data.
- According to this rule mapping of tasks is similar to mapping of required data onto the processes. At this stage we will discuss commonly used techniques of distributing arrays among processes.
- The owner computes rule is most often used in High - Performance Fortran compilation systems.
- According to this rule the required calculation will be performed by the processor that owns the left hand side element. This is described using an example loop shown below in FORTRAN language.

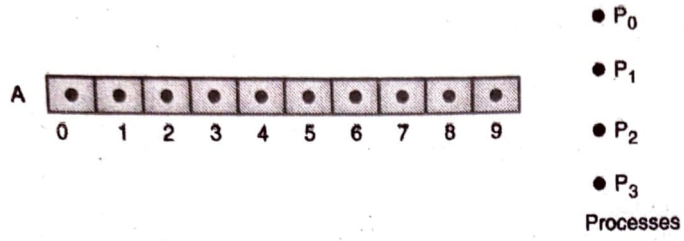
```
DO i = 1,n
a(i-3) = b(i*2)/c(i+j)-a(i**i)
END DO
```

- In this example the expression a(i-3) is the left hand side of the assignment statement. This expression is owned by a processor in a multiprocessor system.
- This induces that the assignment will be performed by the processor owns a(i-3) expression. Therefore all the components present at the right hand side should be made available to the processor for performing computation without any hassle.

2.4.2(B) Block Distributions

- The block distribution deals with the homogeneous distribution of computational load over a regular data structure such as a Cartesian grid.
- According to block distribution blocks of size S of the vector is assigned to processes for mapping of elements during processing.
- This distribution is based on the approach of assigning contiguous portions of the array to different processes.
- In this distribution technique an array with m - dimension is distributed among a set of processes in the fashion such as each process gets contiguous block of array elements.
- The block distribution of array elements is most of the time suitable when the computation performs on the nearby elements of an array called as locality of interaction.
- The block distribution is described in Fig. 2.4.3 with pictorial representation.
- In Fig. 2.4.3 an array named as **A** of size **10** is considered. Also assume four processes P_0, P_1, P_2 and P_3 execute in parallel to compute on array elements.

- In the first case the array elements are distributed to four processes. Here, the block size is assumed three.
- This means processes P_0, P_1 and P_2 will get three elements each of the array A whereas process P_3 will get remaining one element.



• Distribute array elements into processes with block size as three :

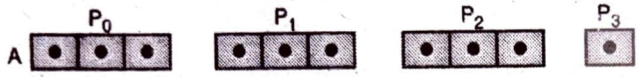


Fig. 2.4.3

- In the first case, the array elements are distributed to four processes. Here, in Fig. 2.4.4, the block size is assumed four.
- This means processes P_0, P_1 will get four elements each of the array A whereas process P_2 will get two elements and all elements have been distributed. The last process P_3 will not get any element.

Distribute array elements into process with block size as four :

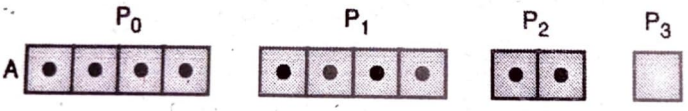


Fig. 2.4.4

- In this scheme, the A[i] element is mapped to the processor $\lfloor i/b \rfloor$ if distribution is BLOCK(b).
- Now, we will consider the distribution of an array A of M elements over P processes. The mapping of the global index m where it lies between 0 to M represented as (0 <= m < M) is handled. The global index m of the data object will be mapped to an index pair (P, I).
- In the current discussion p is used to specify processes to which the elements are mapped.
- The value of p is represented by 0 <= p < P with indication of its range. The value of variable i indicates the location of the element in the array. The mapping represented as m → (p, i) for block distribution is defined as:

m → (floor(m/L), m mod L)

where, L is defined as **L = ceiling(M/P)**.

- The block distribution is described using an example where the parameters p, m etc. are used. Let's assume we have M = 23 data elements to be distributed over three processes (P = 3). Also consider the block size S = 8.

m	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22
p	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	2 2 2 2 2 2 2
I	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6
B	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	2 2 2 2 2 2 2

- It is clear in the example that uneven distribution occurs. Here because of uneven distribution the last block becomes smaller than the others. A global block number B is also shown here for indication of block distribution.

☞ Distribution of M × N matrix using block distribution

- In block distribution of any array each process gets a contiguous part of the array data. An M × N matrix is distributed using two different approaches : **row - wise** and **column - wise distribution**.
- Each process receives N/P rows of a matrix; here P indicates the number of processes used for mapping of tasks when elements are distributed along with row - wise distribution approach.
- On the other hand each process gets M/P columns of the matrix when column - wise distribution approach is applied. The block distribution approach is suitable for some computations like matrix - vector multiplication.

☞ 2-D distribution on processes

- We have to assume a process grid for the distribution of array elements on processes.
- In general one has to assume the size of the process grid, assume now as $P_1 \times P_2$. The number of processes for which this grid can be used is $P = P_1 * P_2$.
- For example consider the distribution of N × N matrix. The matrix can be divided into $N/P_1 \times N/P_2$ sub matrices. Fig. 2.4.5 shows the 4 × 4 process grid.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

Fig. 2.4.5

- This 4×4 process grid is used to execute processes in parallel with distribution of data in the form of the grid.
- Once the data required by a process is provided that process can start execution. In this way all processes operate on their part of the data concurrently.

2.4.2(C) Cyclic Distributions

- In the situation where computational load is distributed in inhomogeneous fashion, cyclic distribution is used. The cyclic distribution is used in such case to improve load balancing.
- The consecutive entries of the global vector are used to assign in successive processes.
- The cyclic distribution of data among processes does not use any concept of block numbers. The mapping m → (p,i) is defined as shown below for cyclic distribution :

m →(m mod P, floor(m/P))

- This scheme of distribution can be used to reduce the problems like the load - imbalance and idling.
- The load imbalances occur when the amount of work is different for different part of a matrix. This can be avoided by using the cyclic or block cyclic distributions.
- The cyclic distribution is described in Fig. 2.4.6 using an example where the parameters p, m etc. are used. Let's assume we have m = 23 data elements to be distributed over three processes (p = 3).

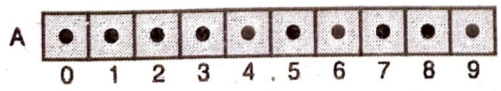
m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
i	0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7

Fig. 2.4.6

☞ Example

1-D Cyclic distribution of array 'A' on four processes

- The Cyclic distribution is described using an example shown in Fig. 2.4.7. Assume there are four processes P_0, P_1, P_2 and P_3 are available so that elements of an array can be mapped onto these for doing computations.



Processes : P_0,P_1,P_2,P_3

Fig. 2.4.7(a)

- Now the cyclic distribution of array 'A' becomes as shown in Fig. 2.4.7(a)

Fig. 2.4.7(b)

In Fig. 2.4.8 the distribution of elements to processes are shown according to cyclic distribution approach. In this example 1-D cyclic distribution on four processes is described.

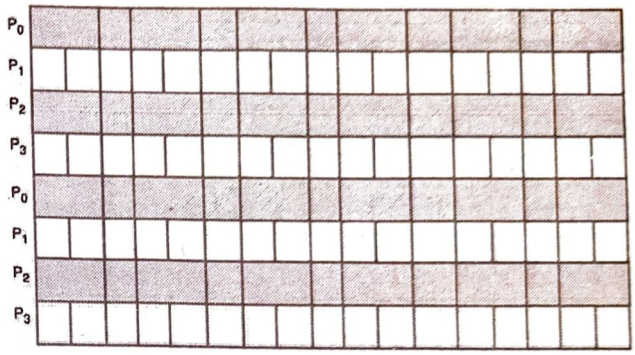


Fig. 2.4.8

2.4.2(D) Block - Cyclic Distribution

- This type of data distribution onto processes is another technique which is slight variation of the block distribution.
- The block - cyclic distribution is the generalization of the block and cyclic distribution.
- In this form of distribution a block is considered which consists of number of consecutive data objects represented by **r**. These consecutive data objects are distributed cyclically onto number of processes represented by **p**.
- The parameters used here are the global index m and index triplet **(p, b, i)**. In this case the global index m lies between **0** and **M** and represented by **m(0<=m<M)**.
- In the index triplet **(p, b, i)** the value of p lies between **0** and **P** and represented as **p(0<=p<P)**.
- The global index of the data object is mapped into index triplet where **p** specifies processes onto which elements are mapped.
- The triplet **(p, b, i)** has p to be specified the process to which data element is mapped. The block number is represented by **b** in process **p** and **i** is the location in the block. The mapping **m → (p, b, i)** in block - cyclic distribution is defined as shown below:

M→(floor((m mod T)/r), floor(m/T), m mod r)

- It is require to know about the block – cyclic distribution is that it reverts to the cyclic distribution in which r = 1 and a block distribution where r = L. Here T = rP
- Consider an example of block - cyclic distribution where M = 23 data objects. These data objects are distributed over three processes (p = 3). The block size represented by r is 2.
- The uneven distribution of data objects are shown in Fig. 2.4.9. This happens because the last block is smaller than others.

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
b	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
I	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
B	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11

Fig. 2.4.9

2.4.3 Randomized Block Distributions

- This type of distribution is considered as a more general form of block distribution. The load balancing is achieved by dividing the array into more number of blocks than the total number of processes currently active for processing the data.
- The load balancing in randomized distribution is similar to a block - cyclic distribution.
- In this technique blocks are distributed randomly among processes but the block distribution is handled uniformly.
- Now consider an example of a one dimensional block distribution in randomized fashion shown in Fig. 2.4.10.
- The vector V with 15 elements included is distributed using randomized block distribution. The assumed block size is 3 and five processes are available.

V = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}

Random(V) = {11, 5, 13, 8, 14, 1, 6, 7, 3, 10, 0, 12, 2, 4, 9}

Now perform mapping of blocks onto processes.

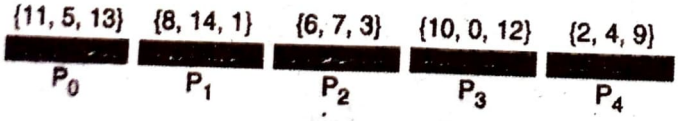


Fig. 2.4.10

2.4.4 Hierarchical Mappings

- There are some algorithms can be represented using Task dependency graphs.
- They are represented in the straight forward manner because of the task implemented on those are naturally suited to represent using Task dependency graphs.
- The mapping in this approach also has some problems like load-imbalance or inadequate concurrency.
- For example hierarchical mapping can be described for the binary tree task dependency graph.
- One such binary tree is shown in Fig. 2.4.11. Here at the top of the tree only few concurrent tasks can be performed because of availability of less number of tasks.
- The decomposition is expected at further level when the level at which number of tasks available are large so that mapping can be handled efficiently.
- In such cases a large task is divided into smaller sub - tasks at further levels.

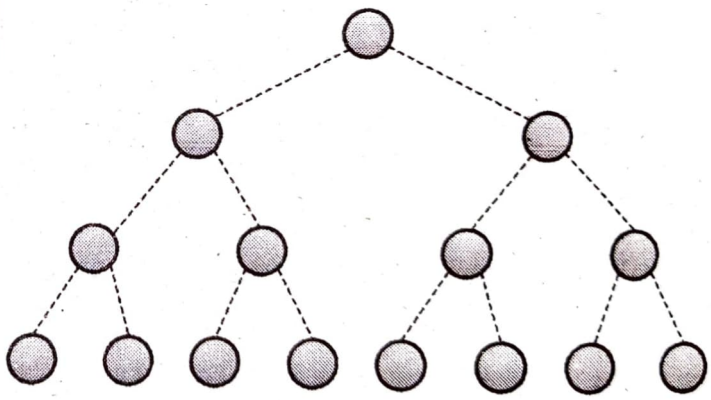


Fig. 2.4.11

- In Fig. 2.4.11, the binary tree is shown with level size as 4.
- Here initially the root task can be divided and assigned into eight processes. In the similar fashion task at the level next to root can be divided and assigned into four processes.
- These divisions at different levels are used for mapping of tasks onto processes and finally at leaf level tasks are mapped with processes in the form of one to one mapping.
- The Fig. 2.4.11 shows the hierarchical mapping based on the descriptions we have just made.

2.4.5 Schemes for Dynamic Mapping

- Dynamic mapping is applied in the situation where static mapping is not efficient because it shows very highly imbalanced distribution of works.
- The dynamic mapping of work distribution to processes used in another situation where task - dependency graph itself is dynamic by nature.
- The dynamic mapping is also called as dynamic load - balancing because it is basically used for balancing of workloads among processes. There are two different classes of dynamic mapping **centralized and distributed**.

1. Dynamic mapping with Centralized Schemes

- In this scheme a common data structure is used centrally to maintain all executable tasks. These tasks can also be maintained centrally by a special process.
- The processes involved in such a system are managed in the master - slave fashion. The master process is a special process responsible to manage the pool of available tasks.
- The other processes referred as slave processes get their work assigned from the master process.
- Any newly generated tasks are stored in the centralized data structure and master process assigns tasks from this structure to slave processes.
- The main advantage of the centralized scheme is its simplicity in implementation.
- It is really easier in terms of implementation as compare to distributed scheme. The limited scalability is the disadvantage of this scheme.
- For example in the case when number of processes has to access central data structure increases; the efficiency of the master process to assign tasks may degrade the overall performance.
- Consider sorting of an example matrix A of size M × N using the centralized mapping approach. The program segment shown below is used to do sorting of entries in each row of A in serial order.

```
for ( i = 0; i < M; i++)
sort( A[i], M );
```

- The sorting of initial elements of an array affects the total sorting time of a particular sorting algorithm.
- This way the time required for sorting of an array elements vary because of dependency of sorting of initial elements.
- This concludes that each iterations of the **'for loop'** used in the program segment considered in this example may take different amount of time.
- In such type of situations load - imbalance may occur when mapping of the task (sorting of an equal number of rows) onto processes.

There are two techniques have been suggested for handling load - imbalance problem are : ***self-scheduling*** and ***chunk scheduling***.

(A) Self-scheduling : This method is used for scheduling of independent iterations of a loop on number of available processes in parallel. This way it may happen that separate iteration of the loop is assigned to different processes for executions.

- All such processes then execute iterations of the loop in the parallel fashion. The approach is based on the use of a central pool of row indices of not yet sorted elements.
- When a process becomes available it picks up an index from the pool and performs sorting on that row. After sorting successfully that index is deleted from the pool of indices. This way all rows are sorted till the pool becomes empty.

(B) Chunk scheduling : The overall task is divided into number of subtasks. A chunk refers to a group of such tasks. In this scheme chunks are created and assigned to different processes for parallel executions. The problem of load - imbalance occurs when the number of tasks assigned in a single step is large.

- The problem becomes more severe when the chunk size increases during program progress. This problem is solved by reducing the chunk size when program progresses further.
- This means when the program execution starts the chunk size remains large but when program further progresses the chunk size also reduced.

2. Dynamic mapping with Distributed Schemes

- In this scheme the set of available executable tasks are distributed among processes for executions.
- The work balance is handled at runtime with exchange of tasks among processes.
- In this scheme it is assumed that communications among processes for sending and receiving work-loads is possible. The main advantage of this scheme is that these methods do not encounter any type of bottleneck.

Syllabus Topic : Methods for Containing Interaction Overheads

2.5 Methods for Containing Interaction Overheads

Q. 2.5.1 Explain Methods for Containing Interaction Overheads. (Refer sections 2.5 to 2.5.4) **(8 Marks)**

- A parallel program performs efficiently when the interactions among the concurrent tasks are efficiently handled. There are many factors associated due to which interaction overhead increases in concurrent tasks.
- Some of the factors which are most important and require attention are: amount of data exchanged during interactions, the frequency of interactions and the spatial and temporal pattern of interactions.
- The general techniques used to reduce the interaction overheads are applicable during the stages when the decomposition and mapping schemes for the algorithms are devised. Some techniques are useful algorithm is being programmed in a given paradigm.

2.5.1 Maximizing Data Locality

- The tasks executed by different processes have to access some common data in most of the parallel programs.
- A particular task is able to compute efficiently when the required data upon which it operates are available in time.
- The data locality technique is used here to make the required data available to the tasks so that task interactions can be reduced.
- The various techniques like increase the reuse of recently accessed data and reduce the data access frequency up to some level so that data locality can be maximized.
- The techniques used in the modern processors for improvement of the data availability through cache, work almost similar like the schemes implemented for improving the data availability.

☞ Minimize Volume of Data-Exchange

- The interaction overhead is reduced using the basic technique in which the overall volume of shared data is minimized.
- In this basic approach the shared data that needs to be accessed by concurrent tasks are targeted to be minimized.
- This approach is based on the facility where the locality of the temporal data is maximized. This way the consecutive references to the same data increases.
- The requirements for bringing more data into local memory or cache are minimized because of performing most of the computations on local data.
- One more approach is considerable here to decrease the amount of shared data required to be accessed by multiple processes is use of local data for intermediate results. This way shared data access is performed only for storing of the final results of the computations.

☞ Minimize Frequency of Interactions

- The overheads occur in the interaction of tasks in parallel programming is reduced by minimizing the frequency of interaction.
- The use of shared data in large pieces through restricting of algorithm may affect the reduction in interaction frequency.
- This way the overall interaction overhead is reduced, even if such restructuring does not necessarily reduce the overall volume of shared data that need to be accessed.

2.5.2 Overlapping Computations with Interactions

- In the parallel program executions different processes involved in the computations have to cooperate and communicate with each other. These processes have to wait shared data to be arrived from other processes.

- The overlapping of computation with the interaction can be handled when a process has to wait for interaction at that time it can be assigned to do other independent computations. There are various approaches used to overlap computations with interactions.
- We can consider an example where overlapping is possible, simply identify the parts of the program code can be executed before initiating the interaction.
- Thereafter the parallel program is required to be structured in such a way that the interaction is to be initiated at an earlier point in the execution than it is needed in the original algorithm.

2.5.3 Replicating Data or Computations

- The interaction overheads amongst the concurrent tasks can also be reduced using the replication of data and computations.
- The explicit programmer intervention is not required for the read - only data needed for frequent access in the shared - address - space paradigm because such data very often affected by the caches.
- The architectures and programming paradigms where read - only access to shared data is expensive and harder as compare to local data access the obvious solution is based on replication of the data. This way parallel programming using message - passing paradigm benefits from the replication of data.
- The message - passing significantly reduces the interaction overheads and simplifies the programming.
- The downside of the data replication is increased memory requirements of a parallel program.
- The overall memory requirement for replicated data storage increases linearly with the number of concurrent processes.
- The direct restriction imposed because of the memory limitation is the limitations on the size of the problem to be solved on a particular parallel machine.

2.5.4 Overlapping Interactions with Other Interactions

- The overlapping of interactions with other interactions amongst several pairs of processes can reduce the effective volume of communications.
- We can consider four processes P_0, P_1, P_2 and P_3 as an example of overlapping interactions in a message - passing paradigm.
- In this paradigm broadcasting of data from process P_0 to all other processes works in the fashion as follows.
- Process P_0 sends data to be sent to process P_2. In the next step the data is transferred to process P_1 and concurrently P_2 sends the data it had received from P_0 to P_3. In this way the operation is completed in two steps instead of multiple steps.

Syllabus Topic : Parallel Algorithm Models

2.6 Parallel Algorithm Models

Q. 2.6.1 Write short note on Parallel Algorithm Models. (Refer section 2.6.1) **(6 Marks)**

- The parallel algorithm models are used to describe the strategy for partitioning data and how these data are processed.
- A model is used to provide appropriate structuring of parallel algorithms on the basis of two techniques : **selection of a partitioning and mapping technique, proper use of strategy for interaction minimization.**
- There are basically six different types of models for parallel algorithms have been suggested by researchers.

2.6.1 The Data-Parallel Model

- This model of algorithm design is considered as one of the simplest models of parallel algorithm design.
- The various tasks identified in the problem to be solved are mapped to processes. The mapping of tasks onto processes are handled statically or semi - statically.
- In this strategy once the data have been distributed parallel operations can be performed. In this way the operations performed by each task are similar but the data on which these operations work are different.
- The parallelism occurs here and it is a form of data parallelism because identical operations are being performed on the different data sets.
- The overall task to be performed may be divided into different phases. The data upon which the tasks work in different phases may be different.

- In this approach different computation phases are interspersed with interactions to synchronize the tasks.
- The problem being solved using this approach is divided among number of tasks on the basis of data partitioning.
- The data partitioning is used because all tasks perform similar computations and a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.
- The two popular paradigms shared - address space and message - passing can be used for implementations of data - parallel algorithms.
- The interaction overheads occur in data parallel model but it can be minimized by adopting some strategy on implementations.
- A locality preserving decomposition is used for the interaction overhead minimization and in the similar way computation and interaction can be overlapped in the design.
- In the data parallelism, one of the key characteristics for most of the problem is that the degree of data parallelism increases with the size of the problem.
- This effectively provides the options for the use of more processes to solve larger problems.
- The parallel matrix multiplication is one of the examples of the data - parallel computations.

2.6.2 The Task Graph Model

- We have already discussed that the task dependency graph is used to describe the computations in any parallel algorithms. The task - dependency graph is explicitly used in mapping of some algorithms.
- The interaction costs among the tasks can be reduced by the use of interrelationships among the tasks in task dependency graph.
- The problems in which tasks are associated with large amount of data as compare to the actual computations can be solved effectively using this model.
- The data movement cost among task is optimized by the use of mapping of tasks to processes statically.
- The paradigm with globally addressable space can be used to share work in more easier ways, but disjoint address space paradigm can also be used because mechanisms are available to share the work.
- The parallelism described with the task - dependency graph where each task is an independent task is called as **task - parallelism**.
- There are several examples of parallel algorithms based on the task - graph model and some of them are **parallel quick - sort**, parallel implementation of algorithms based on **divide and conquer** etc.

2.6.3 The Work Pool Model (or Task Pool Model)

- The mapping of tasks onto processes for parallel execution is described by the dynamic mapping approach so that load balancing can be handled.
- The dynamic mapping is used because according to that any task can be mapped with any process provided that it should have the availability of task assignment.
- The work pool model uses the dynamic mapping approach for task assignments.
- In the work pool model of algorithm design centralized or decentralized mapping can be adopted.
- The work to be performed sometimes available in advanced but can be generated at runtime also. The executing processes generated the work to be performed and these are added into the globally available work pool.
- The work pool model can be used in the message - passing paradigm when the data associated with the tasks is smaller than the computation associated with the tasks.
- This way the tasks can be moved without causing too much overhead due to interactions. The examples of the work - pool model are loop parallelization using chunk scheduling, parallel tree search etc.

2.6.4 The Master Slave Model (Manager - Worker Model)

- This model of parallel algorithm design designates two different types of processes: master and slave.

- In this scheme one or more processes are designated as master and are responsible to coordinate the activities among all slave processes.
- The master process generates work to be performed and assigns it to the number of worker/slave processes.
- The allocation of the tasks depends upon the size of the task. If the size of the task can be estimated on prior basis the master can allocate the tasks to the required processes.
- In another scenario the slave processes can get the pieces of work at different times.
- The assignment of work to the slaves at different times is preferable when the situation where all slaves cannot wait for the availability of tasks from the master on prior basis.
- In some of the scenarios where work have to be done in different phases this model can be used effectively by assigning different slaves to perform tasks at different phases. In this case the master is responsible to synchronize the activities of the slaves after each phase.
- The master - slave model is generally suitable for shared address space and message passing paradigms.
- This is possible because in the master - slave model interaction among the processes are always in two - way fashion.
- The two - fashion is described as the master always knows that it has to provide things to slaves and in the similar way slaves know that they get things from the master.

2.6.5 The Pipeline or Producer Consumer Model

- This model is based on the passing on stream of data through processes arranged in a succession.
- The data flows through successive processes and at that time each process does some operations on it. This is called stream parallelism because simultaneous executions of different programs on data streams are performed.
- Whenever a new data arrive a new task execution initiates by a process in the pipeline. This model is also called as producer - consumer model because a pipeline acts as a chain of producers and consumers. Suppose P_1 and P_2 are two processes associated with a pipeline and P_1 precedes P_2.
- In this situation process P_2 is the consumer process and P_1 becomes the producer because it provides the input stream for process P_2.
- The pipeline can be arranged in a directed graph fashion also instead of strictly on a linear chain fashion.
- The mapping of tasks onto processes in the pipeline model is usually handled using a static mapping approach.

2.6.6 Hybrid Models

The combination of more than one model of parallel algorithm design is applicable in some cases for solving the problem at hand. These models are used at different phases of the parallel algorithm design. The suitability of a particular model in a phase of the algorithm is taken care and accordingly it is used.

Syllabus Topic : The Age of Parallel Processing

2.7 The Age of Parallel Processing

- The parallel computing activities have been adopted by many industries to fulfil their computing requirements of tasks. In modern machines almost all the processors are equipped with the in-built multicore processors.
- The parallel processing facility is provided into many handheld electronic devices such has mobile phones and portable music players.
- The software development field uses many tools for the developments and the design of parallel software applications because of the availability of multiple processing elements in electronic devices.
- The incorporation of parallel processing in various devices makes the devices like cell phones not only used for call making but also other activities like playing music, Internet access etc. are widely adopted.

2.7.1 Central Processing Units and GPUs

- All the program and instructions are executed in order to derive the necessary data in case of CPUs. The advancement in modern day CPUs have allowed it to crunch more numbers than ever before, but the advancement in software technology meant that CPUs are still trying to catch up.

- The load of the CPU is alleviated by the GPU by handling all the advanced computations necessary to project the final display on the monitor.
- Originally, CPUs handle all of the computations and instructions in the whole computer, thus the use of the word 'central'.
- But as technology progressed, it became more advantageous to take out some of the responsibilities from the CPU and have it performed by other microprocessors.
- In the days before GUIs, the screen was simply a small grid with each box having an 8 bit value that corresponds to a character. This was relatively very easy to do for the CPU, but GUIs have greater resolutions with each pixel having a 16 bit or 32 bit color value.
- GPUs where originally developed to render 2D graphics; specifically, to accelerate the drawing of windows in a GUI. But as the need for 3D and faster graphics acceleration grew, the GPU became faster and more specialized in its task.
- GPUs are now generally floating point processors that can easily crunch geometric computations along with texture mapping tasks.
- Most GPUs have implemented MPEG primitives to make enhance the playback of videos; some even have the capability to directly decode HD video data, taking another task away from the CPU.
- Hardware wise, GPUs and CPUs are similar but not identical. If we looked at the very building block of each, the transistors, we can see that most GPUs already rival CPUs in transistor count.
- The specialized nature of GPUs means that it can do its task much faster than a CPU ever can, but it is not able to cover all of the capabilities of the CPU.
- Multiple GPUs can also be employed to achieve a single goal much like the dual core CPUs currently available. ATI's Crossfire and NVidia's SLI allow users to connect two identical GPU's and make them work as one.

Syllabus Topic : Brief History of GPUs, Early GPU

2.7.2 The Rise of GPU Computing

The GPU is used now a day to perform general purpose computations and because of that the computing tasks are performed in efficient ways. The idea of graphics processors is not new in the field of computing but GPU is comparatively a new concepts in computing.

A brief history of GPUs

- The graphic processing field also evolved like clock speed changes in central processing units.
- The graphically driven operating systems like Microsoft Windows entered in the software field in early 1990s and since then the graphics processing became prime concern for processor manufacturing companies.
- The hardware devices require for handling graphics based systems were developed and adopted by the users and programmers.
- At round the same time the company named as Silicon Graphics popularized the use of three dimensional graphics in various markets like government sector and defense applications.
- The OpenGL library is released by Silicon Graphics in 1992 and the programming interfaces for their hardware were opened.
- The intension of releasing OpenGL was based upon the focus that it should be used as a standardized, platform-independent method for writing 3D graphics applications.
- In the mid-90s the demand for 3D graphics based applications increased because of the significant developments in the 3D gaming industry.
- At the same time companies like NVIDIA, ATI Technologies and others started releasing graphics accelerators in affordable price.
- The capabilities of the consumer graphics hardware further pushed after release of NVIDIA's GeForce 256.
- In such systems, the graphics processors were first time used to perform the computations directly and because of that the scope for the developments for visually interesting applications were enhanced.

- The most important achievements in the GPU computing technology is the release of NVIDIA's GeForce 3 series in 2001.
- The GeForce 3 series was the computing industry's first chip to implement Microsoft's then-new DirectX 8.0 standard. This standard required that compliant hardware contain both programmable vertex and programmable pixel shading stages.
- For the first time, developers had some control over the exact computations that would be performed on their GPUs.

❑❑❑