High Performance Computing

(Code : 410250) (Compulsory Subject)

Semester VIII - Computer Engineering (Savitribai Phule Pune University)

Unit II : Parallel Algorithm Design

Syllabus

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 :*** *Data, Task, Work Pool and Master Slave Model,* ***Complexities :*** *Sequential and Parallel Computational Complexity, Anomalies in Parallel Algorithms*

As per Updated Syllabus of (Savitribai Phule Pune University)

TechKnowledge™ Publications

2 Parallel Algorithm Design

Unit - II

Syllabus

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: Data, Task, Work Pool and Master Slave Model, Complexities: Sequential and Parallel Computational Complexity, Anomalies in Parallel Algorithms

2.1 Principles of Parallel Algorithm Design - Preliminaries

- Parallel algorithms are mainly meant for executing the concurrent part of the algorithm simultaneously by different processors. The final output of the different processors are finally collected together and the result is obtained.
- It is easy for some algorithms to be divided into parts that can be concurrently executed, and hence achieve a high level of parallelism. For example, if the numbers from 1 to 100 are to be checked, to find which are prime and which are not. If one processor is doing this operation it will take a long time, as it can check only one number at a time. But if we have 100 processors, we can divide the task amongst 100 processors, each checking one number to be prime or not. Hence this operation can be done concurrently.
- But there are quite a few operations that cannot be done concurrently. In this case, we need to find the concurrent operations, or else the algorithm has to be executed on one of the processors of the so many processors in the multiprocessor system. Most of the operations in the numerical methods are iterative, with each iteration dependent on the previous iteration. Hence they cannot be executed concurrently.

2.1.1 Preliminaries

For designing parallel algorithms the following steps are to be followed :

1. We need to find the pieces of work or tasks that can be done concurrently.
2. Then these tasks are to be mapped onto multiple processors i.e. we need to map the processes vs processors
3. The distribution of input/output & intermediate data across the different processors is also to be considered as this requires a lot of extra time.
4. Thus we also require the management of the accesses of shared data.
5. Finally the synchronization of the processors at various points of the parallel execution is also to be done.

The most important things to be considered while making parallel algorithms is to maximize the concurrency, reduce the overheads due to parallelization and hence maximize the speedup.

2.1.2 Decomposition, Tasks and Dependency Graphs

Q. Explain Granularity, Concurrency, and Dependency Graph. SPPU - Oct. 18 (In Sem.), 6 Marks

Q. Define and explain the term : Granularity. SPPU - May 19, 2 Marks

2.1.2(A) Decomposition

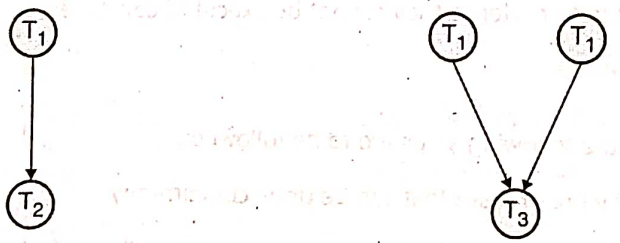
- When we think about how to parallelize a program we use the concepts of decomposition.
- Decomposition is the process of dividing a computation into smaller parts. Some or all parts of this computation may be executed in parallel.
- The number and size of tasks into which a problem is decomposed determines the **granularity** of the decomposition.
- When a computation is divided into a large number of small tasks, it is referred as **fine-grained** decomposition.
- Whereas the **course-grained** decomposition consists of a small number of large tasks.
- To achieve a high degree of concurrency, we should have good decomposition techniques.

2.1.2(B) Tasks

- The first step in developing a parallel algorithm is to decompose the problem into tasks that are candidates for parallel execution.
- Task is an indivisible sequential unit of computation. It is a programmer defined units of computation into which the main computation is subdivided by means of decomposition.

2.1.2(C) Task-Dependency Graphs

- Tasks may have dependencies on other tasks. If the input of task B is dependent on the output of task A, then task B is dependent on task A.
- Task dependency graphs are used to describe the relationship between tasks.
- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next.



(a) Task T_2 is dependent on T_1 (b) Task T_3 is dependent on T_1 and T_2

Fig. 2.1.1

For Example

(i) There is an edge between two tasks T_1 and T_2 if T_2 must be executed after T_1 task. A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other (Refer Fig. 2.1.1).

(ii) Here in graph 1 (Refer Fig. 2.1.2), Task 7 is dependent on

Task 4 and Task 7 , 6, 5,3,2,1.

In dependency graph we have start and finish node.

- Start node- the nodes with no incoming edges
- Finish node- the nodes with no outgoing edges

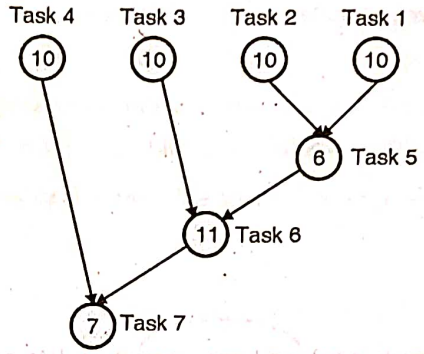


Fig. 2.1.2 : Graph 1

Basic features of task dependency graph

1. **Critical path**
   - Critical path is the longest directed path between any pair of start and finish nodes.
   - The sum of the weights of nodes along this path is the critical path length.

   For graph 1, critical path length is 10 + 6 + 11 + 7 = 34, Node 10 → 6 → 11 → 7

2. **Total amount of work**
   - If assumed that each tasks takes one unit of time then it is referred as total amount of work.
   - Here in graph 1, total amount of work is 10 * 4 + 6 + 11 + 7 = 64. As we have total 7 tasks and we are assuming each one takes 1 unit time.

3. **Maximum degree of concurrency**
   - Maximum number of tasks executed simultaneously in a parallel program is termed as the maximum degree of concurrency.
   - In the above graph 1, maximum degree of concurrency is 4.

4. **Average degree of concurrency**

   The average number of tasks that can run concurrently over the entire duration of execution of the program is termed as average degree of concurrency.

$$\text{Average degree of concurrency} = \frac{\text{Total amount of work}}{\text{Critical path length}}$$

   For graph 1,

$$\text{Average degree of concurrency} = \frac{64}{34} = 1.88$$

5. **Degree of concurrency**

   The number of tasks that can be executed in parallel. The degree of concurrency varies with the granularity of the decomposition.

2.2 Decomposition Techniques

Q. Explain any three data decomposition techniques with example. **SPPU - Oct. 2018 (In Sem.), 4 Marks**

De-composition Techniques are as follows :

1. Data decomposition
2. Recursive decomposition
3. Exploratory decomposition
4. Speculative decomposition

2.2.1 Data Decomposition

- If we want to execute a problem which is having large amount of data parallely then data decomposition is the technique.
- In data decomposition, the data is partitioned across various tasks.
- Two steps are required for decomposition of computation :
  1. The data on which the computations are performed is divided into sub parts.
  2. These sub parts is then executed by different processors concurrently.

For example : Matrix multiplication

- Consider matrix multiplication problem, multiply matrix A and B of size 2*2 which will store results in matrix C.
- The output of matrix C is divided into four tasks as,

$$T_1 = AE + BG$$

$$T_2 = AF + BH$$

$$T_3 = CE + DG$$

$$T_4 = CF + DH$$

- Now each task will be assigned to different four processors P_1, P_2, P_3 and P_4. All these processors will execute these 4 tasks concurrently so that execution will be faster.

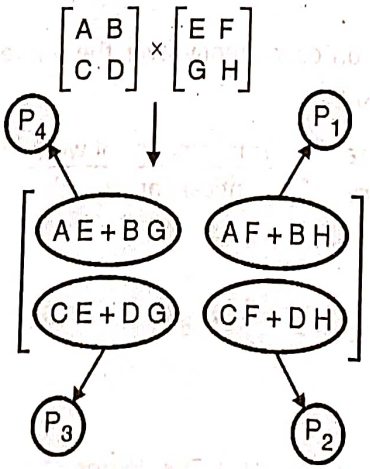


Fig. 2.2.1

Note : In data decomposition, **data is different** with every processor but **computation is same.** As in above example all matrix elements in tasks are different but computation is same for all tasks i.e multiplication and addition.

2.2.2 Recursive Decomposition

- The recursive decomposition is based on providing concurrency in problem that can be solved in divide and conquer strategy.
- In divide and conquer, we follow the approach that to solve the problem directly if it is small and if it is not able to solve then decompose the problem into sub problems and solve the sub problems.
- Each sub problem is solved recursively and then the solution to original problem is obtained by combining the results of these sub problems.

For example : Merge sort

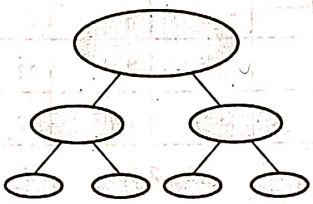


Fig. 2.2.2

2.2.3 Exploratory Decomposition

- Sometimes the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration i.e search of a state space of solutions.
- In other words, your problem is in running state and in the running state it is also decomposed which is referred to as exploratory decomposition.

For example : Puzzle problem

- The puzzle problem is we have to transform from initial state to a desired final state.
- In this problem, there are 4 tiles numbered through 1 to 4 are arranged in 2×2 grid shown in Fig. 2.2.3

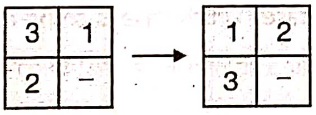


Fig. 2.2.3

- One tile is left blank, so that moves can be made. The 2 possible moves here are up and left with blank tile. So we have divided initial problem into 2 sub problems.
- So here we are searching whether we have reached till final goal and also concurrently applying moves. So we are executing problems at the same time we are decomposing problems into sub tasks.
- The same puzzle problem is possible with 4×4, 8×8, 15×15 grids. Only moves will be more.
- So in exploratory decomposition, computation is split into tasks, each task searching a different portion of the search space.

15 puzzle problem

(a)

1	6	2	4
9	5	3	8
13	↑	7	11
14	10	15	12

(b)

1	6	2	4
9	5	3	8
13	10	7	11
14	→	15	12

(c)

1	6	2	4
9	5	3	8
13	10	7	11
↓	14	15	12

(d)

1	6	2	4
9	5	3	8
↓	10	7	11
13	14	15	12

(e)

1	6	2	4
←	5	3	8
9	10	7	11
13	14	15	12

(f)

1	6	2	4
5	6	7	8
9	10	11	12
13	14	15	←

Fig. 2.2.4

- Fig. 2.2.4(a) shows the initial configuration of puzzle. Different sequence of movements right, up, down and left from initial to final steps are shown in Fig. 2.2.4(b) to Fig. 2.2.4(e). Fig. 2.2.4 (f) shows the final expected configuration.

2.2.4 Speculative Decomposition

- It is impossible to identify independent tasks whenever dependencies between tasks are not known in advance.
- In this decomposition, the action to be taken is based on output of the preceding part.
- This decomposition is used when a program may take one of many possible computationally significant branches depending on the output of the other computations that precede it.

For example : Switch case

- In switch case, among multiple available cases which case is to be considered is based on the input (expression) which has come from its preceding part.

```
Compute expr;
switch(expr)
{
        case 1:  compute t1;
        break;
        case 2 : compute t2;
                 break;
        ------
        -------
}
```

Here lots of options are available to choose, but which one to select is depend on previous preceding code part.

- One more example of speculative decomposition is **topological sorting**.
- In topological sorting we will check incoming edges of every node and if the node doesn't have any incoming edge then only we can add that node in the sorting list.

So here also we are looking for incoming edges (looking the previous state).

2.3 Characteristics of Tasks and Interactions

Once the problem has been decomposed into independent tasks, the characteristics of these tasks can have impact on performance of parallel algorithms.

We know that task is the basic unit of computations.

Characteristics of tasks

1. Generation of tasks

The tasks that in used to implement parallel algorithms are generated as static or dynamic.

(i) Static task generation

- Before computation if all the tasks are known then it is static task generation.
- Data or recursive decomposition often leads to static task generation.
- **For example :** matrix and algorithms, image processing

(ii) Dynamic task generation

- Here tasks are not available a priori. In some cases, the tasks to be executed are created dynamically based on the decomposition of data. Tasks are generated as we perform computation.
- Exploratory and speculative decomposition can generate tasks dynamically.
- **For example :** quick and merge sort, puzzle game.

2. Size of task

- Every task takes some amount of time for its completion.
- Time duration required by the task to complete is termed as a size of task.
- Size will be either uniform or non-uniform.
- Uniform size - all tasks are of the same size. Matrix multiplication decomposition is of uniform size.
- Decomposition of merge sort is example of non-uniform size.

3. Knowledge of task size

We should know about the size of task i.e how much time the particular task will take for completion. Because for mapping task to different processors we should know in advance the size of the task. We will learn about mapping in next session.

4. Data size

Data associated with the task must be available to the process performing the task while mapping. This will avoid excessive data movement overheads.

2.4 Mapping Techniques for Load Balancing

Q. Differentiate Static and Dynamic mapping techniques for load balancing. **SPPU - Dec. 18, May 19, 6 Marks**

- We know that in parallel programming, processes execute a particular task. So **mapping is the technique used to assign tasks to processes.**
- The objective of mapping is that all tasks should complete in the shortest amount of time.
- Mapping techniques are used to solve the major sources of overheads that is
  - Load imbalance(some processes may spend being idle)
  - Inter-process communication
- So assigning a balanced load to each process is necessary. The main objective of load balancing is the amount of computation assigned to each processor is balanced so that some processors do not idle while others are executing tasks.

Techniques of mapping

There are two techniques of mapping for load balancing

1. Static mapping

- Before executing algorithm, tasks are distributed among processes.
- If tasks sizes are unknown, a static mapping can lead to serious load-imbalances.
- Static mapping is applicable for tasks that are generated statically and advance uniform computational requirements are known in advance.

2. Dynamic mapping

- Dynamic mapping is also referred to as dynamic load balancing
- Tasks are distributed among processes at runtime.
- It is applicable for tasks that are,
  - Generated dynamically and
  - If we have non-uniform computational requirements.
- There are two different classes of dynamic mapping
  - (i) Centralized dynamic mapping
  - (ii) Distributed dynamic mapping

(i) Centralized dynamic mapping

- In centralized dynamic mapping, all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes.
  - **Master :** Process mange a group of available tasks.
  - **Slave :** Depend on master to obtain work.
- When a slave process has no work, it takes a portion of available work from master and when a new task is generated, it is added to the pool of tasks in the master process.

- The main problem with centralized dynamic mapping is,
  - When many processes are used, master process may become bottleneck.
- Solution to this problem is,
  - **Chunk scheduling :** every time a process runs out of work it gets a group of tasks from master. In other words, a process will get chunks of tasks in a single step and not a single task.

(ii) Distributed dynamic mapping

- In distributed dynamic mapping, tasks are distributed among processes which exchange tasks at run time to balance work.
- Each process can send or receive work from other processes.
- The main problem with distributed dynamic mapping is that how the sending and receiving processes paired together at run time and who will initiate the work transfer (sending or receiving process).

2.5 Methods for Containing Interaction Overheads

Q. Explain any four methods for containing interaction overheads. **SPPU - Dec. 18, 6 Marks**

- For an efficient algorithm, reducing the interaction overhead among concurrent tasks is important.
- The overhead that a parallel program incurs due to interaction among processes depends on many factors, such as, the volume of data exchanged during interactions, the frequency of interaction, the spatial and temporal pattern of interaction etc.
- General techniques used to reduce the interaction overheads are,
  1. Maximizing data locality
  2. Minimizing Contention and Hot-Spots
  3. Overlapping Computation with interaction
  4. Replicating data or computation
  5. Using Optimized collective interaction operations

2.5.1 Maximizing Data Locality

- For some parallel programs, the different processes require access to common input data or may be processes require data generated by other processes. In this case it will increase interaction overheads.
- The interaction overhead can be reduced by using techniques that promote the use of local data or data that have been recently fetched.

Various schemes of data locality enhancing that try to,

- Minimize the volume of non-local data that are accessed,
- Maximize the reuse of recently accessed data and,
- Minimize the frequency of accesses.

2.5.2 Minimizing Contention and Hot-Spots

- In previous sessions we have learned that interaction overheads can be reduced by reducing frequency and volume of data transfer.
- Though, the data access and inter-task interaction patterns can often lead to contention that can increase the overall interaction overhead.
- A frequent source of contention for shared data structures or communication channel is because of centralized schemes for dynamic mapping.

  Contention occurs when,

  - Multiple tasks try to access the same resources concurrently.
  - Multiple simultaneous transmissions of data over the same interconnection link.
  - Multiple simultaneous accesses to the same memory block.
  - Multiple processes sending messages to the same process at the same time.
- The contention may be reduced by choosing a distributed mapping scheme.

2.5.3 Overlapping Computation with Interaction

- After an interaction has been initiated , the amount of time that processes spend waiting for shared data to arrive or to receive additional work can be reduced by doing some useful computations during this waiting time.
- There are number of techniques that can be used for overlapping computations with interaction.
- The simplest technique is initiating an interaction early enough so that it is completed before it is needed for computation.
- For this, need to identify computations that can be performed before the interaction and do not depend on it.
  - Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original algorithm.
- In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process.
  - Instead of this if the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance.
- On a shared-address-space architecture, the overlapping of computations and interaction is assisted by prefetching hardware.
  - So if the prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed.

2.5.4 Replicating Data or Computation

- Techniques that are used to reduce interaction overheads

Replication of data

- Multiple processes may require frequent read-only access to shared data structure such as hash-table.
- Replicate a copy of the shared data structure on each process.

- After the initial interaction during replication, all subsequent accesses to this data structure are free of any interaction overhead.

Replicating computation

- The processes in a parallel program often share intermediate results in some situations it may be more cost effective for a process to compute these intermediate results than to get them from another process that generates them.
- However, data replication increases the memory requirements of a parallel program as we need to store replicated data on each process.
- So data replication must be used selectively to replicate relatively small amount of data.

2.5.5 Using Optimized Collective Interaction Operations

- The interaction pattern among concurrent activities is often static and regular.
- A class of such static and regular interaction patterns are those that are performed by groups of tasks and they are used to perform certain type of computations on distributed data.
- A need to identify such collective interaction operations that appear frequently in many parallel algorithms to reduce the interaction overhead.
- For example, broadcasting some data to all processes

2.6 Parallel Algorithm Models

An algorithm model is typically a way of structuring a parallel algorithm by selecting decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

There are basically six different types of model :

1. Data parallel model
2. The task graph model
3. The task pool model
4. Master/Slave model
5. Pipeline/producer-consumer problem
6. Hybrid model

2.6.1 Data Parallel Model

- It is simple algorithm model.
- In this model, the tasks are statically mapped onto processes and each task performs similar operations.
- It is a result of identical operations being applied concurrently on different data items, is called data parallelism.
- The work may be done in phases. Data parallel computation phases are interspersed with interactions to synchronize tasks.
- It can be implemented in both shared-address-space and message passing paradigm.
- Interaction overhead can be minimized.

- The degree of data parallelism increases with size of problem.
- It solves large problems effectively.
- **For example :** Dense matrix multiplication.

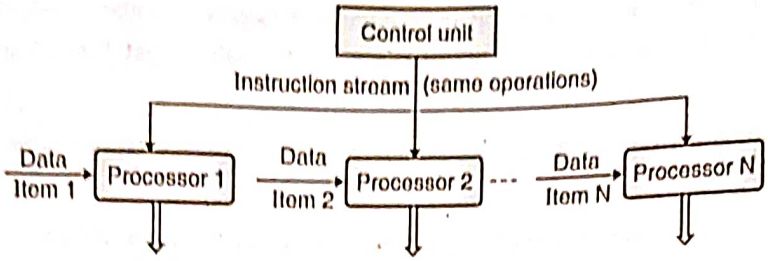


Fig. 2.6.1

2.6.2 Task Graph Model

- Starting from task dependency graph, the interrelationship among tasks are utilized to promote locality or to reduce interaction cost.
- It is used to solve problem in which amount of data associated with tasks is large.
- Tasks are mapped statically to optimize the cost of data.
- For example : Parallel quick sort, sparse matrix factorization
- This type of parallelism that is naturally expressed by independent tasks in task dependency graph called **Task Parallelism.**

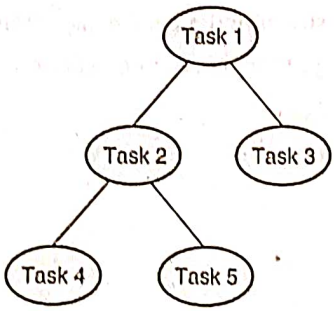


Fig. 2.6.2

2.6.3 The Task Pool Model

- It is also called as work pool model.
- It is characterized by dynamic mapping of tasks onto process for local balancing in which task may be performed by any process.
- No desired premapping.
- Mapping may be centralized or decentralized
- Pointer to task may be stored in physically shared list, queue, hash table or tree.
- Work may be statically available in the beginning or could be dynamically generated.
- Used when amount of data associated with task is relatively small.
- **For example :** Parallelization of loops by chunk scheduling

2.6.4 Master-Slave Model

- It is also called as Manager-Worker model.
- One or more master processes generate work and allocate to worker processes.
- Tasks may be allocated prior if manager can estimate size of task.
- In other scenario, workers are assigned small pieces of work at different times.
- No desired premapping is done, manager will allot work.
- The manager –Worker model can be generalized to hierarchical or multi-level model in which top level manger sends chunks of tasks to low level managers who further sub divides tasks.
- It is suitable to shared-address space or message passing paradigm.
- Master should not become bottleneck, which may happen if task are too small.
- It reduced waiting time.

2.6.5 Pipeline / Producer-Consumer Problem

- In this model, a stream of data is passed on through succession of processes, each of which perform same task.
- This simultaneous execution of different programs on a data stream is called Stream Parallelism.
- Processor could form pipeline in the form of arrays, trees, graphs etc.
- A pipeline is a chain of producer and consumer.
- Each process in pipeline can be viewed as consumer of sequence of data items for process preceding it and producer of data for process following it.
- It involves static mapping.
- Load balancing is function of task granularity.
- For example : Parallel factorization algorithm

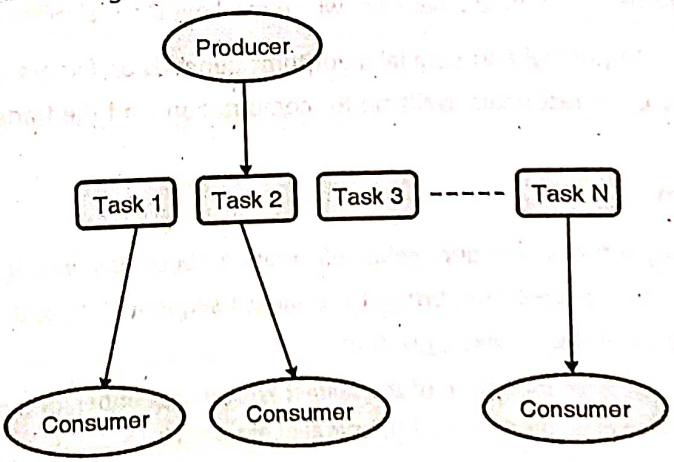


Fig. 2.6.3

2.6.6 Hybrid Model

- More than one model can be applicable to problem.
- It is a composition of multiple models.

- Task dependency graph is used.
- **For example :** Parallel Quicksort

2.7 Complexities

2.7.1 Sequential and Parallel Computational Complexity

- An algorithm's analysis enables us to decide whether or not it is useful. An algorithm is often evaluated based on how long it takes to execute (Time Complexity) and how much space it takes up (Space Complexity).
- Storage space is no longer an issue because sophisticated memory devices are now affordable. Thus, space complexity is not given significant weight.
- Sequential computational complexity refers to the **amount of time** and **resources required** to execute an algorithm on a single processor, while parallel computational complexity refers to the amount of time and resources required to execute the same algorithm on multiple processors simultaneously.
- The complexity of a sequential algorithm is typically measured in terms of its **time complexity**, which is the number of steps required to complete the algorithm as a function of the input size. Commonly used notation for time complexity is Big O notation, which provides an upper bound on the number of steps required as the input size grows.
- In contrast, the complexity of a parallel algorithm is typically measured in terms of its parallel speedup, which is the ratio of the time required to execute the algorithm sequentially to the time required to execute it in parallel.
- The parallel speedup is influenced by factors such as the **number of processors** available, the **communication overhead** between processors, and the **granularity of the algorithm**, which refers to the degree to which it can be decomposed into smaller sub problems that can be executed in parallel.
- In some cases, parallel algorithms may achieve a **speedup** that is proportional to the number of processors available, which is known as linear speedup. However, achieving linear speedup can be challenging due to factors such as communication overhead and load balancing, which can limit the scalability of parallel algorithms.
- Overall, the choice between sequential and parallel algorithms depends on factors such as the size and complexity of the problem being solved, the resources available for computation, and the trade-offs between time and space complexity.

1. Speedup of an Algorithm

- Calculating a parallel algorithm's speedup yields information about how well it performs. Speedup is defined as the ratio of the worst-case execution time of the fastest sequential algorithm for a given problem to the worst-case execution time of the parallel algorithm.

  Speedup = Worst case execution time of the fastest known sequential for a particular problem / Worst case execution time of the parallel algorithm

2. Number of processors used

- When evaluating the effectiveness of a parallel algorithm, the number of processors used is a crucial consideration. The cost of purchasing, operating, and maintaining the computers is calculated. The cost of the solution is directly proportional to the number of processors an algorithm uses to solve a problem.

$$P(n) = O(n)$$

- The number of processors used can have a significant impact on the time complexity of a parallel algorithm. Generally, as the number of processors used increases, the time complexity of the algorithm decreases, since the workload is distributed across multiple processors, allowing for faster execution.
- However, the relationship between the number of processors used and the time complexity of the algorithm can vary depending on the nature of the problem being solved and the algorithm being used. In some cases, increasing the number of processors may not provide a significant improvement in performance, due to factors such as communication overhead or contention for shared resources.
- Therefore, the optimal number of processors to use for a given parallel algorithm will depend on a variety of factors, including the size of the problem being solved, the nature of the algorithm, the characteristics of the hardware being used, and the specific goals of the computation.

3. **Total cost**

- The sum of a parallel algorithm's time complexity and processor count is known as the total cost of that algorithm.

$$\text{Total Cost} = \text{Time complexity} \times \text{Number of processors used}$$

- In addition to the time complexity, the total cost complexity of a parallel algorithm also takes into account the cost of using multiple processors, including the cost of communication and synchronization between processors.
- The total cost complexity of a parallel algorithm is generally measured in terms of the product of the time complexity and the number of processors used, plus any additional costs associated with communication and synchronization.
- Therefore, while increasing the number of processors used can potentially decrease the time complexity of a parallel algorithm, it may also increase the total cost complexity if the additional communication and synchronization costs outweigh the benefits of parallelization.
- As with the time complexity, the optimal number of processors to use for a given parallel algorithm will depend on a variety of factors, including the size of the problem being solved, the nature of the algorithm, the characteristics of the hardware being used, and the specific goals of the computation. A careful analysis of the total cost complexity is necessary to determine the optimal number of processors for a given problem and algorithm.

4. **Work-efficiency**

- It is more crucial to create a parallel algorithm that is efficient than it is asymptotically fast. The total number of operations, or work, that an algorithm completes, determines how effective it is. An algorithm's work and its execution time are the same on a sequential machine. The work on a parallel computer is only the processor-time product. As a result, work W = Pt is produced by an algorithm on a P-processor computer that requires time t. In either scenario, the work, assuming that the cost of a parallel machine is proportional to the number of processors in the machine, closely captures the real cost to complete the computation. If an algorithm performs the same amount of work as the fastest sequential algorithm to within a constant factor, we refer to it as work-efficient (or simply efficient).

$$T^{sequential} \sim T^{parallel} \cdot N^{processors}$$

2.7.2 Anomalies in Parallel Algorithms

- Anomalies in parallel algorithms refer to unexpected or unusual behavior that can occur during the execution of parallel algorithms. These anomalies can be caused by a variety of factors, including hardware failures, race conditions, and synchronization issues.
- Some common types of anomalies in parallel algorithms include:
  1. **Deadlocks :** A deadlock occurs when two or more processes are blocked, waiting for each other to release a resource or complete a task.
  2. **Race conditions :** A race condition occurs when the outcome of an operation depends on the timing or order in which processes are executed.
  3. **Starvation :** Starvation occurs when a process is unable to access a shared resource because other processes are monopolizing it.
  4. **Load imbalance :** Load imbalance occurs when some processes in a parallel algorithm are heavily loaded, while others are idle, leading to reduced efficiency.
  5. **Communication overhead :** Communication overhead refers to the time and resources required to exchange data between parallel processes, which can be a significant bottleneck in some algorithms.
- To avoid these anomalies, it is important to carefully design parallel algorithms, taking into account factors such as load balancing, synchronization, and communication patterns. Additionally, implementing appropriate error-handling mechanisms can help mitigate the impact of hardware failures or other unexpected events that can occur during the execution of parallel algorithms.

1. Deadlocks

- Deadlock is an anomaly that can occur in parallel algorithms when two or more processes or threads are blocked and waiting for each other to release resources that they hold. In a deadlock situation, none of the processes can proceed, and the program becomes stuck in an infinite loop.
- Deadlocks can occur in parallel algorithms due to various reasons, such as improper resource allocation, incorrect locking, or insufficient synchronization. For example, in a parallel algorithm where multiple processes access a shared resource, a deadlock can occur if two or more processes hold the resource and are waiting for other processes to release it.
- To prevent deadlocks, it is essential to carefully design parallel algorithms to ensure proper synchronization and resource allocation. One common approach is to use locks or semaphores to enforce mutual exclusion and prevent multiple processes from accessing the same resource simultaneously. However, improper use of locks or semaphores can also lead to deadlocks.
- To avoid deadlocks, it is also important to ensure that processes release resources when they are no longer needed and to design algorithms with a clear and consistent flow of control to avoid circular dependencies between processes. Careful testing and debugging can also help identify and resolve potential deadlocks before they occur in a production environment.

2. Race conditions

- A race condition is another type of anomaly that can occur in parallel algorithms when two or more processes or threads access a shared resource simultaneously, resulting in unpredictable or incorrect behavior.

- In a race condition, the outcome of the program depends on the order in which processes or threads access the shared resource. This can lead to inconsistent or incorrect results, as well as unexpected behavior, such as crashes or data corruption.
- Race conditions can occur in parallel algorithms due to various reasons, such as improper synchronization or timing, incorrect use of locks or semaphores, or inadequate testing and debugging.
- To prevent race conditions, it is essential to carefully design parallel algorithms to ensure proper synchronization and timing. This can include using locks or semaphores to enforce mutual exclusion, or using other synchronization techniques, such as barriers or condition variables, to coordinate the behavior of processes or threads.
- Race conditions can be difficult to detect and reproduce, as they are dependent on the specific timing and scheduling of the threads or processes involved. To prevent race conditions in parallel algorithms, programmers often use synchronization primitives such as locks, semaphores, or barriers to coordinate access to shared resources and ensure that only one thread or process can access a resource at a time. Additionally, careful design of the algorithm to minimize shared resource access can also help to prevent race conditions.

3. Starvation

- In parallel algorithms, starvation refers to a situation where some processes or threads are unable to make progress because they are not given enough access to the shared resources that they need to complete their tasks. Starvation can occur when multiple processes or threads are competing for limited resources such as CPU time, memory, I/O devices, or network bandwidth.
- The starvation anomaly in parallel algorithms refers to a situation where a process or thread is unable to complete its task due to insufficient access to resources, even though there are enough resources available overall. This can happen when some processes or threads are prioritized over others, causing the lower-priority processes or threads to be blocked or delayed indefinitely
- Starvation can have serious consequences for parallel algorithms, as it can lead to performance degradation, increased latency, and even deadlock in some cases. To prevent starvation, parallel algorithms often use techniques such as resource allocation policies, priority scheduling, and fair queuing to ensure that all processes or threads have fair access to resources. Additionally, it is important to carefully design the parallel algorithm to minimize resource contention and avoid unnecessary delays or blocking of processes or threads.

4. Load imbalance

- In parallel algorithms, load imbalance refers to a situation where the workload is not evenly distributed among the processing elements, such as threads, processes, or nodes. This can happen due to a variety of reasons, such as uneven input data, variations in the complexity of different parts of the algorithm, or differences in the processing power or memory of the processing elements.
- Load imbalance can lead to poor performance and inefficiency in parallel algorithms, as some processing elements may finish their work much earlier than others and be idle while waiting for the slower processing elements to catch up. This idle time can lead to wasted resources, increased latency, and decreased overall throughput of the algorithm.
- To prevent load imbalance in parallel algorithms, programmers often use load balancing techniques to redistribute the workload among the processing elements dynamically.
- This can be done through various methods, such as task stealing, where idle processing elements take on additional tasks from busy ones, or task migration, where tasks are moved from overloaded processing elements to idle ones.

- Load balancing can help to ensure that all processing elements are utilized efficiently and that the algorithm completes in a timely manner. Additionally, careful design of the algorithm to minimize load imbalance, such as dividing the workload into smaller, equal-sized chunks, can also help to prevent load imbalance.

5. Communication Overhead

- In parallel algorithms, communication overhead refers to the additional time and resources required for the processing elements to exchange data and synchronize their actions. This overhead can arise due to the need to transfer data between processing elements, coordinate access to shared resources, or maintain consistency across different processing elements.
- Communication overhead can have a significant impact on the performance and efficiency of parallel algorithms, as it can lead to increased latency, decreased throughput, and wasted resources. This is particularly true for algorithms that involve a large amount of inter-process or inter-thread communication, or those that require frequent synchronization among the processing elements.
- To minimize communication overhead in parallel algorithms, programmers often use techniques such as message passing, shared memory, or a combination of both. Message passing involves sending and receiving data between processing elements explicitly, while shared memory allows processing elements to access a common memory space directly. Careful design of the algorithm can also help to minimize communication overhead by reducing the amount and frequency of data transfers, and by reducing the need for synchronization among processing elements.
- It is important to note that while minimizing communication overhead can improve the performance of parallel algorithms, it should not be done at the expense of correctness or scalability. Achieving a balance between minimizing communication overhead and maintaining the correctness and scalability of the algorithm is key to achieving optimal performance.

Review Questions

Q. 1	Explain Principles of Parallel Algorithm Design.	(6 Marks)
Q. 2	Explain the different methods for Containing Interaction Overheads.	(4 Marks)
Q. 3	Explain Parallel Algorithm Models.	(6 Marks)
Q. 4	What are the characteristics of Tasks and Interactions ?	(4 Marks)
Q. 5	Explain Decomposition, Tasks and Dependency Graphs.	(6 Marks)
Q. 6	Describe in brief Data-decomposition and Recursive decomposition.	(4 Marks)
Q. 7	What are the Characteristics of Tasks?	(4 Marks)
Q. 8	What is Mapping Techniques for Load Balancing?	(4 Marks)
Q. 9	Describe in detail for Methods for Containing Interaction Overheads.	(6 Marks)
Q. 10	Explain in detail Parallel Algorithm Models.	(6 Marks)
Q. 11	Explain performance parameters of sequential and parallel algorithms	(6 Marks)
Q. 12	Explain different anomalies in parallel algorithms.	(6 Marks)

□□□