Advanced Computing and Big Data

Abdullah H. Hassan

Hansel Setiadi

Acknowledgments

This work was prepared by:

Abdullah H. Hassan
2106759880

Hansel Setiadi
2206014744

As a final requirement for *Advanced Computing and Big Data* course, which is one of the master's degree courses in Mathematics at the University of Indonesia.

Thanks to:

Prof. Alhadi B., S.Si., M.Kom., Ph.D.
Dr. Risman Adnan, M.Si.

Table of Contents

CHAPTER 1: INTRODUCTION TO PARALLEL ALGORITHM

A computer is a machine that can be programmed to carry out sequences of arithmetic or logical operations (computation) automatically. The first computer was found in the early 19th century, with just a concept to aid in navigational calculations. Meanwhile the first proper digital computer machine was invented at pre-20th century by the U.S. Navy, which was in the form of a small analog computer to use aboard a submarine.

A parallel algorithm is a type of algorithm that is designed to be executed on a parallel computing system, which consists of multiple processors or computing resources that can work together to solve a problem. Parallel algorithms are used to solve problems that are too large or complex to be solved efficiently on a single processor, and they are often used in scientific and engineering applications where large amounts of data need to be processed and analyzed.

There are several types of parallel algorithms, including data parallel algorithms, task parallel algorithms, and pipeline parallel algorithms. Data parallel algorithms operate on large data sets by dividing the data into smaller chunks and applying the same operation to each chunk in parallel. Task parallel algorithms divide a problem into smaller tasks that can be executed independently, and pipeline parallel algorithms divide a sequence of operations into stages that can be executed in parallel.

Designing parallel algorithms involves developing strategies and techniques for efficiently solving problems using multiple processors or computing resources. There are several design techniques that have been developed to support the design of parallel algorithms, including divide and conquer, data parallelism, task parallelism, pipeline parallelism, and load balancing.

Performance measurements for parallel algorithms are used to evaluate the efficiency and effectiveness of parallel algorithms in terms of their execution time, throughput, and other metrics. Some common performance measurements for parallel algorithms include speedup, efficiency, and scalability. It is important to carefully consider the performance measurements that are relevant to a particular parallel algorithm and computing environment.

Overall, parallel algorithms play a critical role in the field of computer science and are essential for solving many of the large and complex problems faced by society today.

1.1. Sequential vs Parallel Algorithms

In computer science, a sequential algorithm is an algorithm that is executed in a specific order, usually from top to bottom. A sequential algorithm typically follows a linear flow of control, meaning that it executes one step at a time, in the order that the steps are written. Sequential algorithms are typically easier to design and implement, but they may be slower to execute than parallel algorithms.

On the other hand, a parallel algorithm is an algorithm that can be executed concurrently, meaning that multiple parts of the algorithm can be executed at the same time. Parallel algorithms are typically more complex to design and implement, but they can be much faster to execute than sequential algorithms, especially on computers with multiple processors or cores.

One way to classify algorithms is by the amount of concurrency they use. An algorithm that uses no concurrency at all is called a sequential algorithm, while an algorithm that uses full concurrency is called a fully parallel algorithm. Algorithms that use some concurrency, but not full concurrency, are called partially parallel algorithms.

There are many different approaches to designing and implementing parallel algorithms, including data parallelism, task parallelism, and pipeline parallelism. Each approach has its own trade-offs and benefits, and the best approach for a given problem depends on the characteristics of the problem and the hardware on which the algorithm will be executed.

1.2. Why Use Parallel Computing?

There are several reasons why parallel computing can be useful:

1. Speed: Parallel algorithms can be much faster to execute than sequential algorithms, especially on computers with multiple processors or cores. This is because parallel algorithms can exploit the hardware's parallelism to execute multiple parts of the algorithm at the same time, whereas sequential algorithms can only execute one step at a time.
2. Scalability: Parallel algorithms can scale up to larger problem sizes more easily than sequential algorithms. This is because parallel algorithms can take advantage of additional hardware resources (e.g., additional processors or cores) to solve larger problems, whereas sequential algorithms are limited by the speed of a single processor or core.
3. Efficiency: Parallel algorithms can be more efficient than sequential algorithms in terms of resource usage. This is because parallel algorithms can divide the work of a problem across multiple processors or cores, reducing the overall workload of each individual processor or core. This can lead to better utilization of hardware resources and lower energy consumption.
4. Complexity: Parallel algorithms can allow you to solve problems that would be too complex for a sequential algorithm to handle. This is because parallel algorithms can divide a large problem into smaller, independent parts that can be solved concurrently, reducing the overall complexity of the problem.

Overall, parallel computing can be an effective way to solve certain types of problems more quickly and efficiently, especially on computers with multiple processors or cores. However, it is important to carefully consider the trade-offs and challenges of parallel computing, as it can also introduce additional complexity and overhead.

1.3. A Simple Example of Parallel Computing: Solving linear systems of equations in parallel

The steps for solving the linear system of equations using Gaussian elimination, with all equations:

1. Divide the system of equations into smaller subproblems: The first step in parallelizing Gaussian elimination is to divide the system of equations into smaller subproblems that can be solved concurrently. This can be done by dividing the system into rows or columns, or by using a block-partitioning approach. For example, suppose we want to divide the system into two subproblems, each containing two equations:

$$ax + by + cz + \ldots = d$$
$$ex + fy + gz + \ldots = h$$

Subproblem 1:

$$ax + by + cz + \ldots = d$$
$$ex + fy + gz + \ldots = h$$

Subproblem 2:

2. Solve the subproblems concurrently: Once the system of equations has been divided into smaller subproblems, each subproblem can be solved concurrently using a standard sequential Gaussian elimination algorithm. For example, suppose we want to eliminate the x variable from the first subproblem:

Subproblem 1:

$$ax + by + cz + \ldots - d = 0$$

$$(ex + fy + gz + \ldots - h) + \left(\frac{a}{e}\right) * (ex + fy + gz + \ldots - h) = 0$$

This results in a new system of equations with one fewer variable:

$$\left(1 - \left(\frac{a}{e}\right)\right) * ex + fy + gz + \ldots - h = 0$$

$$by + cz + \ldots = d - \left(\frac{a}{e}\right) * h$$

3. Combine the solutions: After the subproblems have been solved, the solutions must be combined to obtain the final solution to the original system of equations. This can be done by combining the solutions in a way that preserve the dependencies between the subproblems. For example, suppose we have solved the first subproblem and obtained a solution for the y and z variables:

$$y_1 = \frac{d - \left(\frac{a}{e}\right) * h}{b}$$
$$z_1 = \ldots$$

And suppose we have solved the second subproblem and obtained a solution for the x and y variables:

$$x_2 = \ldots$$
$$y_2 = \ldots$$

To obtain the final solution, we can substitute the values of y1 and y2 into the equations of the original system:

$$ax + by_1 + cz + \ldots = d$$
$$ex + fy_2 + gz + \ldots = h$$

This results in a new system of equations with one fewer variable:

$$ax + by_1 + cz + \ldots = d$$
$$ex + fy_2 + gz + \ldots = h$$

4. Check for convergence: It is important to check for convergence when solving iterative methods in parallel. If the solutions are not converging, it may be necessary to adjust the convergence criteria or to use a different parallelization strategy. For example, suppose we are using an iterative method to solve the linear system of equations, and we are checking for convergence by comparing the difference between successive iterations to a tolerance value:

```
if (abs(x_new-x_old) < tolerance && abs(y_new-y_old) < tolerance && abs(z_new-z_old) <
tolerance) {
// the solution has converged
}
else {
// the solution has not converged, continue iterating
}
```

It is important to note that parallelizing Gaussian elimination can be challenging, and it requires a good understanding of parallel programming concepts and techniques. It is also important to carefully consider the trade-offs and challenges of parallel computing, such as the overhead of communication and synchronization between processors or cores, and the potential for race conditions and other types of bugs.

1.4. *Flynn’s Taxonomy*

The most popular classification of parallel algorithms is Flynn’s Taxonomy. Flynn's taxonomy is a classification system for computer architectures based on the way they handle different types of computation. The taxonomy is named after Michael J. Flynn, who proposed it in 1966.

Flynn's taxonomy consists of four categories:

1. Single Instruction Single Data (SISD): This category represents a traditional, sequential computer architecture that executes a single instruction on a single data item at a time. SISD architectures are the simplest and most common type of computer architecture.
2. Single Instruction Multiple Data (SIMD): This category represents a computer architecture that executes a single instruction on multiple data items at the same time. SIMD architectures are typically used to accelerate the execution of certain types of algorithms, such as those that involve processing large amounts of data in parallel.
3. Multiple Instruction Single Data (MISD): This category represents a computer architecture that executes multiple instructions on a single data item at the same time. MISD architectures are relatively rare and are typically used for specialized applications, such as error-correcting code or fault-tolerant computing.
4. Multiple Instruction Multiple Data (MIMD): This category represents a computer architecture that executes multiple instructions on multiple data items at the same time. MIMD architectures are the most flexible and powerful type of computer architecture, and they are often used for parallel computing applications.

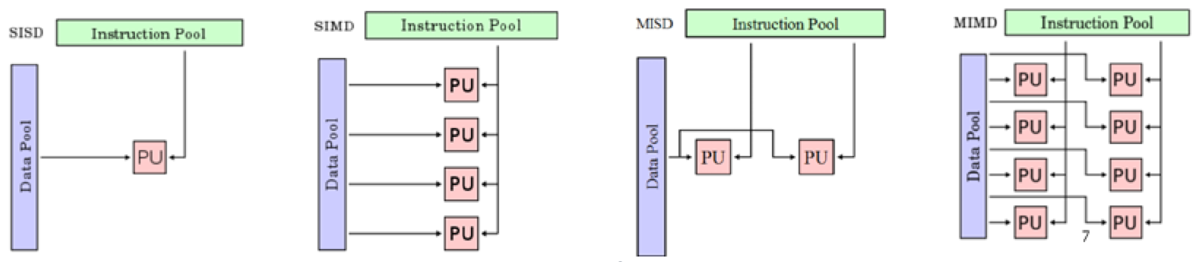


Figure 1: Categories of Flynn's Taxonomy

Flynn's taxonomy is a useful tool for understanding the differences between different types of computer architectures and their capabilities. It is important to note, however, that the boundaries between the categories are not always clear-cut, and many modern computer architectures do not fit neatly into one of the four categories.

CHAPTER 2: PERFORMANCE MEASUREMENTS

We already know from the previous chapter that parallel algorithm was created to make complex computations much faster. But how do we really know scientifically or mathematically that the performance is better?

Performance measurements for parallel algorithms are used to evaluate the efficiency and effectiveness of parallel algorithms in terms of their execution time, throughput, and other metrics. Some common performance measurements for parallel algorithms include speedup, which measures the improvement in performance achieved by using multiple processors or computing resources; efficiency, which measures the fraction of the total available parallelism that is utilized by the algorithm; and scalability, which measures the ability of the algorithm to maintain good performance as the number of processors or computing resources is increased.

Other performance measurements for parallel algorithms include communication overhead, memory usage, and power consumption. It is important to carefully consider the performance measurements that are relevant to a particular parallel algorithm and computing environment, as different performance metrics may be more or less important depending on the specific problem and application.

2.1. Time Complexity

Definition	:	Time complexity is a measure of the efficiency of an algorithm, which describes how the running time of the algorithm increases as the size of the input increases.
Algorithm	:	Time complexity is typically expressed using big O notation, which is a way of expressing the upper bound on the running time of an algorithm. For example, an algorithm with a time complexity of $O(n)$ means that the running time of the algorithm is directly proportional to the size of the input (n). An algorithm with a time complexity of $O(n^2)$ means that the running time of the algorithm is proportional to the square of the size of the input.
Purpose	:	Time complexity is an important measure of the performance of an algorithm because it can help us understand how the algorithm will scale as the size of the input increases. This is especially important when working with large data sets or when the input size is expected to grow over time.

Time complexity can be classified into several categories, depending on the rate at which the running time of the algorithm increases as the size of the input increases. Here are some common categories of time complexity:

1. Worst-case complexity: Worst-case complexity refers to the maximum running time of an algorithm, given the worst-case input data. This is the most conservative measure of an algorithm's performance because it assumes that the input data is the most difficult to process. For example, the worst-case complexity of a sorting algorithm is the time it takes to sort the worst-case input data (i.e., the data that is already sorted in the reverse order).
2. Average-case complexity: Average-case complexity refers to the expected running time of an algorithm, given a random input data. This is a more realistic measure of an algorithm's performance because it considers the fact that the input data may not always be the worst-case. For example, the average-case complexity of a sorting algorithm is the time it takes to sort a randomly shuffled list of numbers.
3. Best-case complexity: Best-case complexity refers to the minimum running time of an algorithm, given the best-case input data. This is the least conservative measure of an algorithm's performance because it assumes that the input data is the easiest to process. For

example, the best-case complexity of a sorting algorithm is the time it takes to sort a already sorted list of numbers.

It is important to note that time complexity is a theoretical measure of performance, and it does not always accurately reflect the actual running time of an algorithm on a particular hardware platform. Other factors, such as the hardware and software environment, the specific input data, and the implementation of the algorithm, can also impact the running time of an algorithm.

Example of time complexity: Insertion Sort

Insertion sort is a sorting algorithm that works by iterating through the input data, comparing each element to the elements that come before it, and inserting it into the correct position in the sorted list. The time complexity of insertion sort depends on the input data and the size of the input.

The running time of insertion sort is determined by the number of comparisons and swaps that are required to sort the input data. In the worst case, when the input data is already sorted in the reverse order, the running time of insertion sort is O(n^2), because the algorithm must compare and swap each element with all the elements that come before it. In the best case, when the input data is already sorted, the running time of insertion sort is O(n), because the algorithm only needs to iterate through the input data without making any comparisons or swaps.

Application example for **worst-case** running time:

input: [5, 4, 3, 2, 1] (already sorted in reverse order)
output: [1, 2, 3, 4, 5]

1. Initialize the sorted list to be empty.
2. Iterate through the input list, one element at a time.
3. For each element, insert it into the correct position in the sorted list by comparing it to the elements that come before it.
4. After all elements have been processed, the sorted list is complete.

In this example, the input list is [5, 4, 3, 2, 1] and the output list is [1, 2, 3, 4, 5]. The running time of the insertion sort algorithm in this example is $O(n^2)$, because the input data is already sorted in the reverse order and the algorithm must compare and swap each element with all the elements that come before it.

Application example for **best-case** running time:

input: [1, 2, 3, 4, 5] (already sorted)
output: [1, 2, 3, 4, 5]

1. Initialize the sorted list to be empty.
2. Iterate through the input list, one element at a time.
3. For each element, insert it into the correct position in the sorted list by comparing it to the elements that come before it.
4. After all elements have been processed, the sorted list is complete.

In this example, the input list is [1, 2, 3, 4, 5] and the output list is [1, 2, 3, 4, 5]. The running time of the insertion sort algorithm in this example is $O(n)$, because the input data is already sorted, and the algorithm only needs to iterate through the input data without making any comparisons or swaps.

The time complexity of insertion sort can be analyzed by considering the number of comparisons and swaps that are required to sort the input data. In the worst case, when the input data is already sorted in the reverse order, the time complexity is $O(n^2)$, because the algorithm must compare and swap each element with all the elements that come before it. In the best case,

when the input data is already sorted, the time complexity is $O(n)$, because the algorithm only needs to iterate through the input data without making any comparisons or swaps.
Algorithm: Here is the pseudo-code for the insertion sort algorithm:

```
for i = 1 to n-1
j = i
while j > 0 and a[j-1] > a[j]
swap a[j] and a[j-1]
j = j - 1
```

This algorithm iterates through the input data (a) one element at a time, starting from the second element ($i = 1$). For each element, it compares it to the elements that come before it ($j > 0 \, and \, a[j-1] > a[j]$) and inserts it into the correct position in the sorted list by swapping it with the element that comes before it (swap a[j] and a[j-1]). This process is repeated until the element is in the correct position ($j = j - 1$) or until it reaches the beginning of the list ($j > 0$).

It is important to note that the time complexity of the insertion sort algorithm depends on the input data and the size of the input. In the worst case, when the input data is already sorted in the reverse order, the time complexity is $O(n^2)$, because the algorithm must compare and swap each element with all the elements that come before it. In the best case, when the input data is already sorted, the time complexity is $O(n)$, because the algorithm only needs to iterate through the input data without making any comparisons or swaps. In the average case, the time complexity is also $O(n^2)$, because the input data is randomly shuffled, and the algorithm must perform a similar number of comparisons and swaps as in the worst case.

2.2. Asymptotic Analysis

Asymptotic analysis is a technique used to analyze the performance of algorithms and to compare the efficiency of different algorithms. It is based on the idea of asymptotic notation, which is a way of expressing the upper bound on the running time of an algorithm as the size of the input grows indefinitely. Asymptotic analysis is an important tool in computer science, because it allows us to understand how the performance of an algorithm scales with the size of the input and to predict the behavior of an algorithm for large input sizes.

Asymptotic Notation

There are several asymptotic notations that are commonly used in asymptotic analysis, including big O notation, big omega notation, and big theta notation. These notations are used to express the upper and lower bounds on the running time of an algorithm, as well as the average running time of an algorithm.

Big O notation	**:**	A mathematical notation that is used to describe the upper bound on the running time of an algorithm. It is written as $O(f(n))$, where $f(n)$ is a function that describes the growth rate of the running time as the size of the input (n) increases.
Big omega notation	**:**	A mathematical notation used to describe the lower bound on the running time of an algorithm. It is used to express the minimum amount of time that an algorithm will take to complete, given a certain input size.
Big theta notation	**:**	A mathematical notation used to describe the average running time of an algorithm. It is used to express the expected amount of time that an algorithm will take to complete, given a certain input size.

The function $f(n)$ is usually a mathematical function, such as a polynomial or an exponential function, that describes the upper bound on the running time of the algorithm.

For example, an algorithm with a time complexity of $O(n^2)$ means that the running time of the algorithm is proportional to the square of the size of the input. This means that the running time of the algorithm will increase at a rate of n^2 as the size of the input increases. An algorithm with a time complexity of $\Omega(n)$ means that the running time of the algorithm is at least proportional to the size of the input. This means that the running time of the algorithm will increase at a rate of at least n as the size of the input increases. While an algorithm with a time complexity of $\Theta(n)$ means that the running time of the algorithm is both upper bounded by n and lower bounded by n. This means that the running time of the algorithm will increase at a rate of at least n and at most n as the size of the input increases.

Big omega notation is often used in combination with big O notation to express the average running time of an algorithm. For example, an algorithm with a time complexity of $\Theta(n)$ has an average running time that is both $O(n)$ and $\Omega(n)$, meaning that the running time of the algorithm is both upper bounded by n and lower bounded by n.

Like big O notation, big omega notation is used to simplify and abstract away from the specific details of an algorithm, and to focus on the overall growth rate of the running time as the size of the input increases. It is an important tool in the analysis of algorithms, as it allows us to understand the minimum performance that can be expected from an algorithm and to compare the efficiency of different algorithms.

Big theta notation is often used in combination with big O notation and big omega notation to express the average running time of an algorithm. For example, an algorithm with a time complexity of $\Theta(n)$ has an average running time that is both $O(n)$ and $\Omega(n)$, meaning that the running time of the algorithm is both upper bounded by n and lower bounded by n.

Like big O notation and big omega notation, big theta notation is used to simplify and abstract away from the specific details of an algorithm, and to focus on the overall growth rate of the running time as the size of the input increases. It is an important tool in the analysis of algorithms, as it allows us to understand the expected performance of an algorithm and to compare the efficiency of different algorithms.

Properties of Asymptotic Notation

There are several properties of asymptotic notation that are important to understand when analyzing the performance of algorithms:

1. **Transitivity**

   The property of transitivity states that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. In other words, if an algorithm has a time complexity of O(g(n)), and $g(n)$ has a time complexity of $O(h(n))$, then the algorithm has a time complexity of $O(h(n))$.

   For example, suppose we have the following functions:

   $$f(n) = n^2$$
   $$g(n) = 2n^2 + 3n + 4$$
   $$h(n) = n^3$$

   Since $f(n) = O(n^2)$ and $g(n) = O(n^2)$, we can say that $f(n) = O(g(n))$.

   Furthermore, since $g(n) = O(h(n))$, we can say that $f(n) = O(h(n))$ by transitivity.

2. **Additivity**

The property of additivity states that if $f(n) = O(g(n))$ and $h(n) = O(k(n))$, then $f(n) + h(n) = O(g(n) + k(n))$. In other words, if an algorithm has a time complexity of O(g(n)), and another algorithm has a time complexity of O(k(n)), then the combined running time of the two algorithms has a time complexity of O(g(n) + k(n)). For example, suppose we have the following functions:

$$f(n) = n^2$$
$$g(n) = 2n^2 + 3n + 4$$
$$h(n) = n^3 \, k(n) = 2n^3 + 3n^2 + 4n + 5$$

Since $f(n) = O(g(n))$ and $h(n) = O(k(n))$, we can use the property of additivity to conclude that $f(n) + h(n) = O(g(n) + k(n))$. This means that the combined running time of the two algorithms has a time complexity of $O((2n^2 + 3n + 4) + (2n^3 + 3n^2 + 4n + 5))$.

By understanding these properties of asymptotic notation, we can more accurately analyze the performance of algorithms and compare the efficiency of different algorithms.

2.3. Speed-up of an Algorithm

In parallel computing, the speed-up of an algorithm refers to the improvement in running time that is achieved by executing the algorithm concurrently on multiple processors or machines. The speed-up of an algorithm in parallel computing can be calculated using the following formula:

$$speedup = \frac{T_{seq}}{T_{par}}$$

where T_{seq} is the running time of the algorithm when executed sequentially on a single processor or machine, and T_{par} is the running time of the algorithm when executed concurrently on multiple processors or machines.

For example, suppose we have an algorithm that takes 100 seconds to complete when executed sequentially on a single processor, and we want to know the speed-up that can be achieved by executing the algorithm concurrently on 4 processors. If the running time of the algorithm when executed concurrently on 4 processors is 25 seconds, then the speed-up of the algorithm can be calculated as follows:

$$speedup = \frac{100\, seconds}{25\, seconds} = 4$$

This means that the algorithm achieves a 4x speed-up when executed concurrently on 4 processors. It's important to note that the speed-up of an algorithm in parallel computing is limited by the inherent parallelism of the algorithm and the efficiency of the parallel implementation. In other words, the speed-up that can be achieved is dependent on the amount of work that can be parallelized and the overhead of executing the algorithm concurrently on multiple processors.

By understanding this formula for calculating the speed-up of an algorithm in parallel computing, we can design and implement efficient parallel algorithms and achieve better performance on multi-core or distributed systems.

2.4. Efficiency of an Algorithm

In parallel computing, the efficiency of an algorithm refers to the proportion of the running time that is spent on useful work, as opposed to overhead such as communication and synchronization between processors. The efficiency of an algorithm in parallel computing can be calculated using the following formula:

$$efficiency = \frac{T_{useful}}{T_{total}}$$

where T_{useful} is the running time of the algorithm spent on useful work, and T_{total} is the total running time of the algorithm including both useful work and overhead.

For example, suppose we have an algorithm that takes 100 seconds to complete when executed sequentially on a single processor, and we want to know the efficiency of the algorithm when executed concurrently on 4 processors. If the running time of the algorithm when executed concurrently on 4 processors is 25 seconds, but 5 seconds of this time is spent on overhead such as communication and synchronization, then the efficiency of the algorithm can be calculated as follows:

$$efficiency = \frac{T_{useful}}{T_{total}} = \frac{(25\, seconds - 5\, seconds)}{25\, seconds} = 0.8\, or\, 80\%$$

This means that the algorithm has an efficiency of 80% when executed concurrently on 4 processors, with 20% of the running time being spent on overhead.

It's important to note that the efficiency of an algorithm in parallel computing is dependent on the inherent parallelism of the algorithm and the efficiency of the parallel implementation. In other words, the efficiency of the algorithm is influenced by the amount of work that can be parallelized and the overhead of executing the algorithm concurrently on multiple processors.

By understanding this formula for calculating the efficiency of an algorithm in parallel computing, we can design and implement efficient parallel algorithms and achieve better performance on multi-core or distributed systems.

2.5. Total Cost

In parallel computing, the total cost of an algorithm refers to the overall cost of executing the algorithm, including both the running time of the algorithm and the cost of any resources required to execute the algorithm. The total cost of an algorithm in parallel computing can be calculated using the following formula:

$$total\, cost = T_{total} + C_{resources}$$

where T_{total} is the total running time of the algorithm including both useful work and overhead, and $C_{resources}$ is the cost of any resources required to execute the algorithm, such as the cost of hardware or software licenses.

For example, suppose we have an algorithm that takes 100 seconds to complete when executed sequentially on a single processor, and we want to know the total cost of executing the algorithm concurrently on 4 processors. If the running time of the algorithm when executed concurrently on 4 processors is 25 seconds, but 5 seconds of this time is spent on overhead such as communication and synchronization, and the cost of the hardware and software required to execute the algorithm concurrently is \$1000, then the total cost of the algorithm can be calculated as follows:

$$total\, cost = T_{total} + C_{resources} = (25\, seconds - 5\, seconds) + \$1000 = \$1000$$

This means that the total cost of executing the algorithm concurrently on 4 processors is \$1000.

It's important to note that the total cost of an algorithm in parallel computing is dependent on the inherent parallelism of the algorithm and the efficiency of the parallel implementation, as well as the cost of the resources required to execute the algorithm. By understanding this formula for calculating the total cost of an algorithm in parallel computing, we can design and implement efficient parallel algorithms and choose cost-effective resources to achieve better performance on multi-core or distributed systems.

CHAPTER 3: MODELS OF PARALLEL ALGORITHMS

The model of a parallel algorithm is developed by considering a strategy for dividing the data and processing method and applying a suitable strategy to reduce interactions. There are several models of parallel algorithms that have been proposed and studied in the field of computer science, including the data parallel model, the task graph model, and the work pool model, ... etc.

3.1. Data Parallel Model

Data parallelism is a type of parallel algorithm in which a model is trained on multiple machines, each with a different subset of the data. The model parameters are shared across all machines, and gradients are calculated on each machine and then averaged across all machines to update the model parameters.

This approach is often used when the data is too large to fit on a single machine, or when the training process is too slow on a single machine. It can also be used to train models on distributed hardware, such as multiple GPUs or a cluster of machines.

There are a few different ways to implement data parallelism, such as using data parallelism libraries like Horovod or using distributed training frameworks like TensorFlow's tf.distribute.Strategy API.

One key challenge in data parallelism is ensuring that the gradients are correctly averaged across all machines. This can be achieved by using an all-reduce algorithm, which combines the gradients from all machines and then broadcasts the result back to all machines.

Another challenge is managing the communication between machines, as the model parameters and gradients must be exchanged between machines during training. This can be resource-intensive and may require careful optimization to ensure efficient training.

Example of Data parallel model: Dense matrix multiplication

Consider two matrices A and B with dimensions (m, n) and (n, p) respectively. We want to compute the matrix product $C = A * B$, where C has dimensions (m, p). We can divide the matrices into smaller sub-matrices and use data parallelism to compute the matrix product in parallel. For example, if we have 4 machines, we can divide the matrices into 4 sub-matrices each with dimensions $\left(\frac{m}{4}, \frac{n}{4}\right)$, $\left(\frac{n}{4}, \frac{p}{4}\right)$, and $\left(\frac{m}{4}, \frac{p}{4}\right)$.

The following diagram illustrates the process:

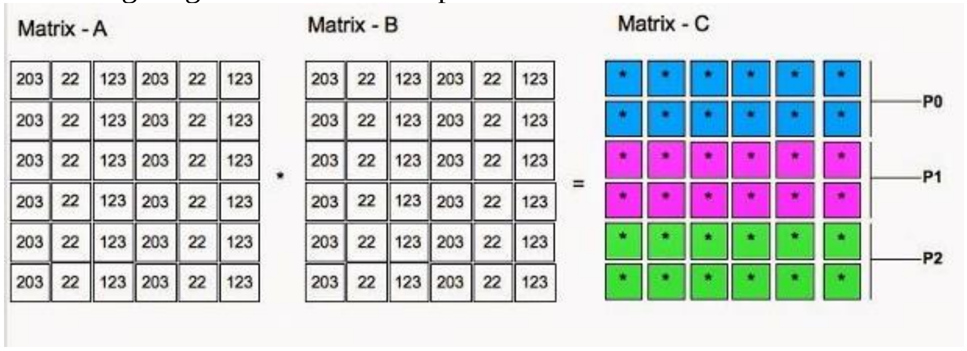


Figure 2: Data Parallel Model - Dense Matrix Multiplication

1. Each machine loads its own sub-matrices of A and B.
2. The machines compute the product of their sub-matrices and store the result in a sub-matrix of C.
3. The sub-matrices of C from all machines are combined to form the result C.

This approach can be useful when the matrices are too large to fit on a single machine and we want to use multiple machines to speed up the computation.

3.2. Task Graph Model

In the context of parallel algorithms, a task graph is a directed acyclic graph (DAG) that represents the dependencies between tasks. Each node in the graph represents a task, and the edges between nodes represent the dependencies between tasks.

Task parallelism is a type of parallel algorithm in which different tasks are executed concurrently. This can be contrasted with data parallelism, in which the same operation is applied to different data concurrently.

Using a task graph allows us to specify the dependencies between tasks and execute them concurrently if possible. This can be useful for optimizing the performance of a workflow or pipeline by identifying tasks that can be executed in parallel.

For example, consider a machine learning pipeline that consists of the following tasks:

1. Preprocess the data
2. Train a model on the preprocessed data
3. Evaluate the model on a test set
4. Visualize the results

We can represent this pipeline as a task graph with the following structure:

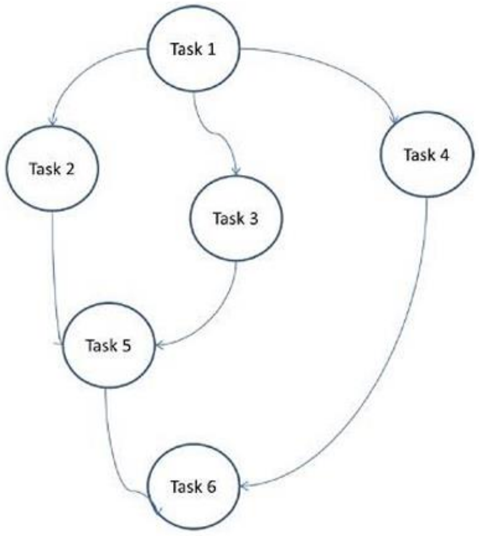


Figure 3: Task Graph Model Structure

In this example, the tasks are represented by nodes in the graph, and the edges between the nodes represent the dependencies between the tasks. For example, the "Train model" task depends on the "Preprocess data" task, and the "Evaluate model" task depends on both the "Train model" and "Preprocess data" tasks.

Using a task graph allows us to specify the dependencies between tasks and execute them concurrently if possible. For example, if the "Preprocess data" and "Train model" tasks can be executed concurrently, they can be assigned to different machines or threads to speed up the overall pipeline.

3.3. Work Pool Model

The work pool model is a type of parallel algorithm in which a pool of worker threads is used to perform tasks concurrently. The tasks are added to a queue, and the worker threads take tasks from the queue and execute them.

This model is often used when there are many independent tasks that need to be executed concurrently. It can be implemented using a thread pool, which is a group of worker threads that are managed by a thread pool manager.

Example − Parallel tree search

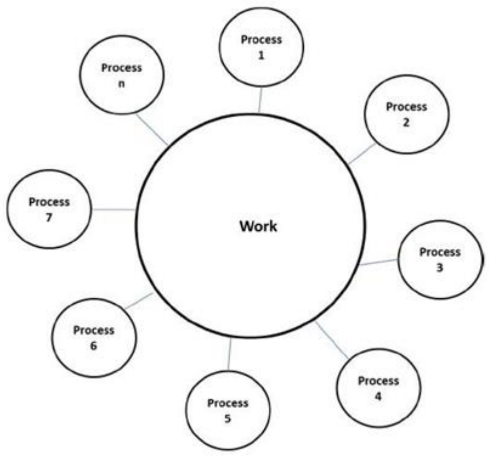


Figure 4: Work Pool Model

Imagine we have a tree with many nodes, and we want to search for a particular node using a breadth-first search (BFS) algorithm. We can use the work pool model to parallelize the search by dividing the tree into smaller sub-trees and assigning each sub-tree to a worker thread for searching.

3.4. Master-Slave Model

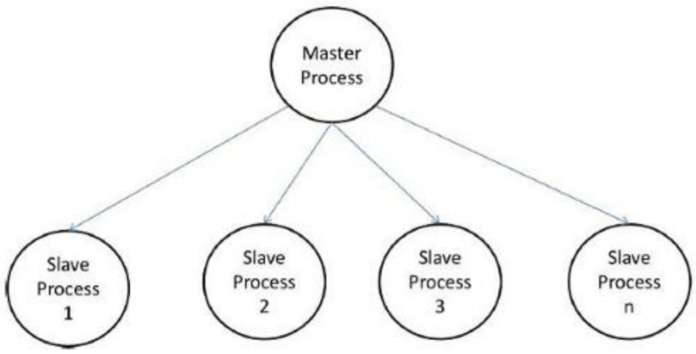


Figure 5: Master-Slave Model

The master-slave model is a type of parallel algorithm in which a central "master" process or thread controls and coordinates the work of multiple "slave" processes or threads. The slaves perform the actual computation, while the master manages the distribution of work and communication between the slaves.

This model is often used when the tasks to be parallelized can be divided into smaller, independent units of work that can be executed concurrently. The master process assigns the tasks to the slaves, and the slaves execute the tasks and return the results to the master.

3.5. Pipeline Model

The pipeline model is a type of parallel algorithm in which the input is processed through a series of stages or "pipelines," each of which performs a specific task. The stages are connected in a pipeline, with the output of each stage serving as the input to the next stage.

This model is often used for data processing or transformation tasks, as it allows the input to be processed in parallel by different stages of the pipeline. Each stage can be implemented as a separate process or thread, and the stages can be executed concurrently if there is no dependency between them.

Here is an example of how the pipeline model can be used to represent an automobile assembly line:

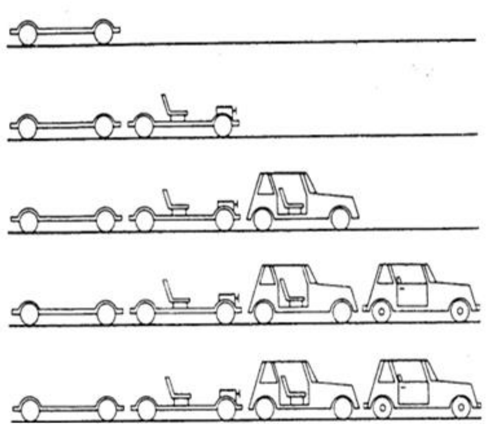


Figure 6: Pipeline Model - Automobile Assembly Line

In this example, the input is a chassis, and the output is a finished automobile. The assembly line consists of multiple stages, each of which performs a specific task, such as installing the engine, adding the body panels, or attaching the wheels.

The input chassis is passed through the assembly line from left to right, and each stage of the pipeline performs its task on the chassis as it passes through. The finished automobile is produced at the end of the pipeline.

The pipeline model can be implemented using a series of threads or processes, with each stage of the pipeline implemented as a separate thread or process. The input chassis is passed between the threads or processes using shared memory or message passing, and the stages can be executed concurrently if there is no dependency between them.

This approach allows the assembly line to operate efficiently and improve the overall speed of production.

CHAPTER 4: DATA STRUCTURE

When we talk about parallel computing, we also need to consider about the data structure. How is it made, what is the data flow that is working on the algorithms. A data structure is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Different types of data structures are suited to different kinds of applications, and some common data structures include arrays, linked lists, stacks, queues, trees, and graphs. Data structures are an important aspect of computer science and are often studied in the context of algorithms, which are used to manipulate the data stored in these structures.

4.1. Linked List

A linked list is a linear data structure in which each element is a separate object, called a node. Each node has a data field and a reference (a link) to the next node in the list. The last node in the list has a reference to null, indicating the end of the list.

Linked lists are useful when you need to insert or delete elements from the middle of a list, because you only need to update the links of the surrounding nodes. However, they are not as efficient as arrays for accessing elements randomly, because you need to follow the links from the beginning of the list to get to a specific element.

There are three main types of linked lists:

1. **Singly linked list**: In a singly linked list, each node has a reference to the next node in the list, but not to the previous one. This means you can only traverse the list in one direction, from the head (the first node) to the tail (the last node).

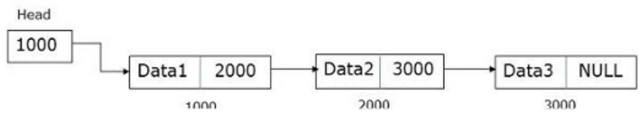


Figure 7: Singly Linked List

2. **Doubly linked list**: In a doubly linked list, each node has a reference to both the next node and the previous one. This allows you to traverse the list in both directions, from the head to the tail, and from the tail to the head.

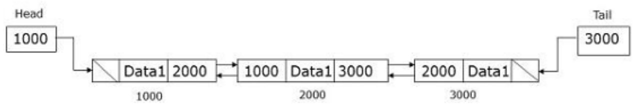


Figure 8: Doubly Linked List

3. **Circular linked list**: In a circular linked list, the last node in the list has a reference to the first node, so the list forms a loop. This allows you to traverse the list continuously, without reaching the end.

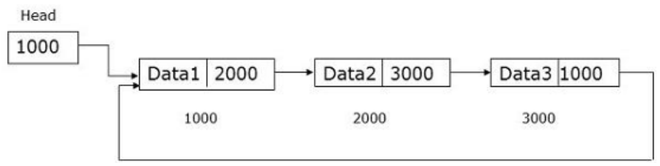


Figure 9: Circular Linked LIst

READ WRITE MEMORY

1. Exclusive read exclusive write (EREW) — every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (CREW) — multiple processors can read a memory cell but only one can write at a time
3. Exclusive read concurrent write (ERCW) — never considered
4. Concurrent read concurrent write (CRCW) — multiple processors can read and write. A CRCW PRAM is sometimes called a concurrent randomaccess machine.

4.2. Array

An array is a data structure that stores a fixed-size sequential collection of elements of the same type. The elements are stored in contiguous memory locations, and they can be accessed by their indices, which are integers that start from 0.

Arrays are useful when you need to store and access many elements efficiently, because you can access any element in constant time ($O(1)$) by its index. However, inserting or deleting elements from the middle of an array can be inefficient, because you may need to shift all the subsequent elements to make room for the new element or close the gap left by the deleted element. Arrays can be created statically or dynamically.

Static array	An array whose size is fixed at compile-time, and their memory is allocated in the stack. They are created by declaring an array with a specific size and initializing its elements.
Dynamic array	Arrays whose size can be changed at runtime, and their memory is allocated in the heap. They are created by allocating memory dynamically using a memory allocator (such as malloc in C or new in C++).

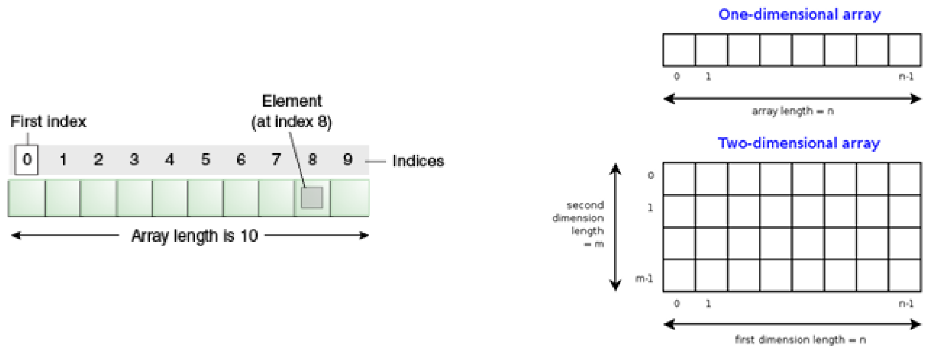


Figure 10: Array

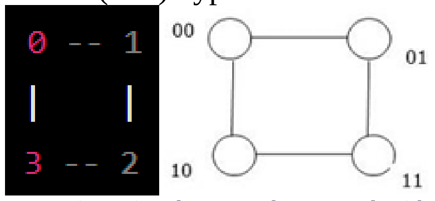
4.3. Hypercube Network

A hypercube network, also known as a cube-connected cycles (CCC) network, is a type of interconnection network used to connect computers in a parallel computing system. It is based on the concept of a hypercube, a geometric shape in which each vertex (corner) is connected to 2^d other vertices, where d is the dimension of the hypercube.

In a hypercube network, each computer is represented by a node, and the nodes are connected by edges. The nodes are arranged in a d-dimensional grid, where d is the dimension of the hypercube. Each node is connected to its 2^d neighbors in the grid, forming a network of 2^d nodes in total.

The advantage of a hypercube network is that it allows for fast communication between any two nodes, because the distance between any two nodes is at most d (the dimension of the

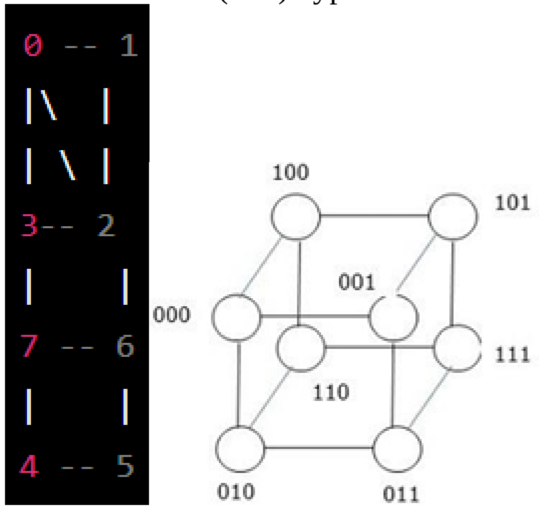
hypercube). This makes it suitable for parallel computing applications that require high-bandwidth communication between nodes.
Here is an example of a 2-dimensional (d=2) hypercube network with 4 nodes:



```
0 -- 1
|    |
3 -- 2
```

Figure 11: 2-Dimensional Hypercube Network with 4 Nodes

And here is an example of a 3-dimensional (d=3) hypercube network with 8 nodes:



```
0 -- 1
|\   |
| \  |
3-- 2
|    |
7 -- 6
|    |
4 -- 5
```

Figure 12: 3-Dimensional Hypercube Network with 8 Nodes

Hypercube networks can be extended to higher dimensions by adding more nodes and edges. However, the number of nodes and edges increases exponentially with the dimension of the hypercube, making it impractical for large-scale systems.

All-to-all communication on Hypercube:

- All-to-all communication refers to the process of sending a message from every node to every other node in the network.
- There are several ways to implement all-to-all communication on a hypercube network, including flooding, topological sorting, and hypercube routing.
- Flooding is a simple and straightforward method, but it can be inefficient because it generates a lot of traffic and may result in multiple copies of the same message being forwarded.
- Topological sorting reduces the number of messages sent and received, but it requires additional overhead to compute the topological sort and may not be suitable for dynamic networks.
- Hypercube routing allows for efficient communication between any two nodes, but it requires additional overhead to compute the routes and may not be suitable for dynamic networks.
- All-to-all communication on a hypercube network can be a resource-intensive operation, and it may not be feasible for large-scale systems with a high number of nodes. In such cases, alternative communication patterns, such as point-to-point communication or collective communication, may be more suitable.

CHAPTER 5: DESIGN TECHNIQUES OF PARALLEL ALGORITHM

Design techniques for parallel algorithms involve strategies for efficiently solving problems using multiple processors or computing resources. These techniques include divide and conquer, data parallelism, task parallelism, pipeline parallelism, and load balancing. Each technique is suited to different types of problems and computing environments, and it is important to consider the characteristics of the problem and the available resources when choosing a design technique.

5.1. Decomposition

Decomposition is a technique used in the design of parallel algorithms to divide a large problem into smaller, more manageable subproblems that can be solved concurrently. The goal is to break down the problem into independent pieces that can be solved simultaneously on different processors or machines, thus increasing the speed and efficiency of the overall computation.

There are several different types of decomposition techniques that can be used in the design of parallel algorithms, including:

Data decomposition	This involves dividing the data to be processed into smaller chunks that can be distributed among the available processors. This can be done in several ways, including by partitioning the data into equal-sized blocks, or by dividing the data based on some logical criteria.
Functional decomposition	This involves breaking down the problem into smaller, independent tasks that can be assigned to different processors. This can be done by identifying the individual steps or stages involved in solving the problem and assigning each step to a separate processor.
Domain decomposition	This involves dividing the computational domain (i.e., the space over which the problem is defined) into smaller subdomains and assigning each subdomain to a separate processor.

Decomposition is a key technique in the design of parallel algorithms, as it allows the problem to be divided into smaller, more manageable pieces that can be solved concurrently. This can greatly increase the speed and efficiency of the overall computation, and is an important consideration in many scientific and engineering applications where large-scale computations are required

Example: dense Matrix-vector multiplication

A dense matrix-vector multiplication is a common operation in scientific and engineering applications and can be performed in parallel using decomposition techniques. Here is an example of how data decomposition can be used to parallelize a dense matrix-vector multiplication:

Suppose we have a matrix A of size 4 x 4 and a vector x of size 4, and we want to compute the product y = Ax. We also have 2 processors available. We can divide the matrix A and vector x into 2 blocks each, as shown below:

$$\begin{aligned} A1 &= [a_{11}\, a_{12}]\, [a_{21}\, a_{22}] \\ A2 &= [a_{23}\, a_{24}]\, [a_{33}\, a_{34}] \\ x1 &= [x_1]\, [x_2] \\ x2 &= [x_3]\, [x_4] \end{aligned}$$

Each processor can then compute the product of its assigned block of the matrix with its assigned block of the vector, resulting in the following partial products:

$$y_1 = A_1 * x1 = [a_{11}x_1 + a_{12}x_2]\,[a_{21}x_1 + a_{22}x_2]$$
$$y_2 = A_2 * x2 = [a_{23}\,x_3 + a_{24}x_4]\,[a_{33}x_3 + a_{34}x_4]$$

The result y can then be obtained by combining the partial products:

$$y = y_1 + y_2 = [a_{11}x_1 + a_{12}x_2 + a_{23}x_3 + a_{24}x_4]\,[a_{21}x_1 + a_{22}x_2 + a_{33}x_3 + a_{34}x_4]$$

This approach allows the matrix-vector multiplication to be performed in parallel, with each processor working on a smaller portion of the computation. This can greatly increase the speed and efficiency of the overall computation, particularly for large matrices and vectors.

5.2. Dependency Graph

In the design of parallel algorithms, dependency graphs can be used to analyze the dependencies between different tasks or elements of the computation, and to identify opportunities for parallelization.

For example, consider a problem that involves the computation of a complex function $f(x)$ for a range of input values x. The function may be composed of several smaller subfunctions, each of which depends on the results of the other subfunctions.

A dependency graph can be used to represent the relationships between the subfunctions and the input values, and to identify which subfunctions are independent and can be computed concurrently. This can help to identify opportunities for parallelization and to design an efficient parallel algorithm.

In a dependency graph for a parallel algorithm, the nodes represent the tasks or elements of the computation, and the edges represent the dependencies between them. The goal is to identify tasks that are independent and can be computed concurrently, and to assign these tasks to different processors or machines.

For example, suppose the function f(x) is composed of subfunctions $f_1(x)$, $f_2(x)$, and $f_3(x)$, as shown in the dependency graph below:

$$f_1(x) \to f_2(x) \to f_3(x)$$

In this case, the subfunction$f_2(x)$ depends on the result of $f_1(x)$, and the subfunction $f_3(x)$ depends on the result of $f_2(x)$.This means that $f_1(x)$ must be computed before $f_2(x)$, and $f_2(x)$must be computed before $f_3(x)$.

However, the subfunctions $f_1(x)$ and $f_3(x)$ are independent and can be computed concurrently. This means that the computation of $f(x)$ can be parallelized by assigning $f_1(x)$ and $f_3(x)$ to different processors and computing $f_2(x)$ sequentially after the results of $f_1(x)$ and $f_3(x)$ have been obtained.

Dependency graphs can be a useful tool in the design of parallel algorithms, as they provide a visual representation of the dependencies between different tasks or elements of the computation and can help to identify opportunities for parallelization.

Example: Database query processing

Database query processing is a good example of how dependency graphs can be used in the design of parallel algorithms.

In database query processing, a dependency graph can be used to represent the relationships between the different operations involved in the query, such as table joins, filters, and aggregations. The goal is to identify tasks that are independent and can be computed concurrently, and to assign these tasks to different processors or machines.

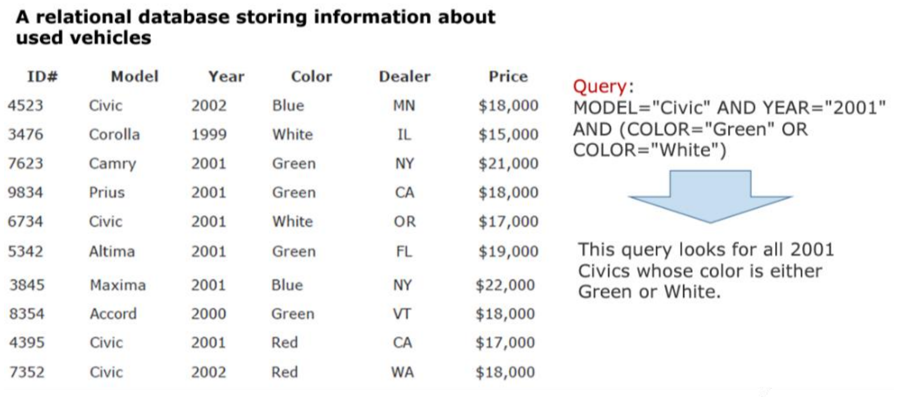
For example, consider a query that involves joining two tables T and R, and applying several filters and aggregations to the resulting data. A dependency graph for this query might look like this:

T ---> join with R ---> filter ---> aggregate

In this case, the join operation depends on the data in both tables T and R, and the filter and aggregate operations depend on the result of the joint operation. However, the data in table T and the data in table R are independent, and the join operation can be parallelized by dividing the data in each table into blocks and assigning each block to a separate processor. The processors can then perform the join concurrently, resulting in a partial join result for each block. The partial results can then be combined to obtain the result of the joint operation.

This approach allows the query processing to be performed in parallel, with each processor working on a smaller portion of the computation. This can greatly increase the speed and efficiency of the overall query, particularly for large tables and complex queries.

Dependency graphs can be a useful tool in the design of parallel algorithms for database query processing, as they provide a clear and visual representation of the dependencies between different operations and can help to identify opportunities for parallelization.



A relational database storing information about used vehicles

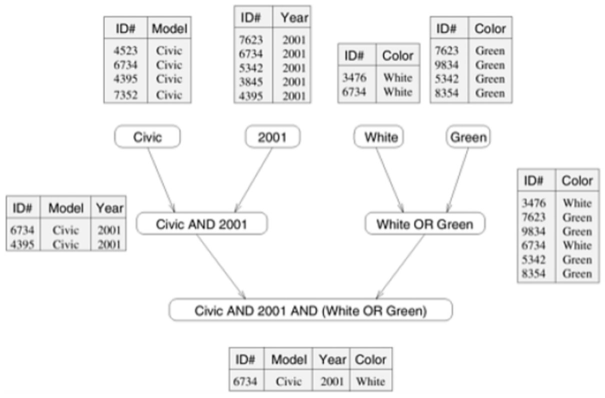
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Query:
MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")

This query looks for all 2001 Civics whose color is either Green or White.

Figure 13: Relational Database Example

The different tables and their dependencies in a query processing operation.



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

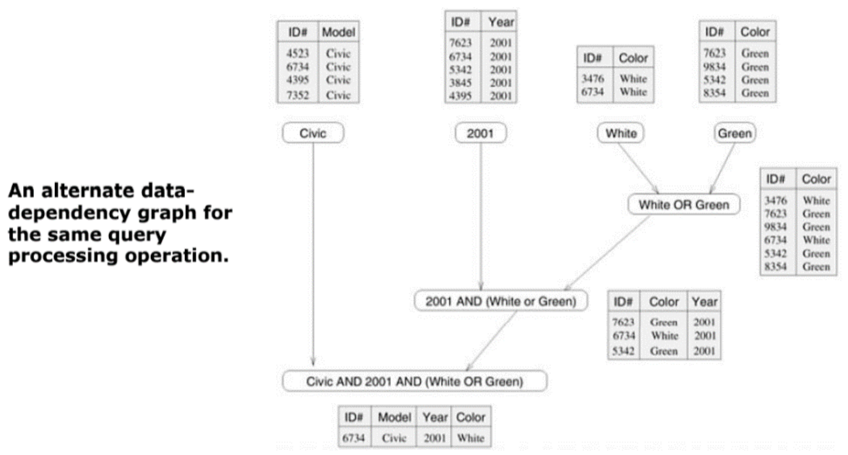
ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

Figure 14: Table Dependency


An alternate data-dependency graph for the same query processing operation.
Civic
2001
White
Green
White OR Green
2001 AND (White or Green)
Civic AND 2001 AND (White OR Green)

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

Figure 15: Alternate Data-Dependency

5.3. Granularity

Granularity refers to the size or complexity of the tasks or units of work in a parallel algorithm. In the design of parallel algorithms, granularity is an important consideration, as it can have a significant impact on the efficiency and performance of the algorithm. There are two main types of granularities: fine-grained and coarse-grained.

Fine-grained granularity refers to algorithms that involve very small or simple tasks that can be executed quickly and independently. Fine-grained algorithms can be highly parallelizable, as each task can be assigned to a separate processor or machine and executed concurrently. However, they may also require a large amount of overhead to manage the tasks and coordinate the parallel execution, which can reduce overall performance.

Coarse-grained granularity refers to algorithms that involve larger or more complex tasks that take longer to execute. Coarse-grained algorithms may require less overhead to manage and coordinate the tasks, but may also be less parallelizable, as the larger tasks may not be easily divided into smaller, independent units of work.

The choice of granularity in a parallel algorithm is often a trade-off between parallelizability and overhead. Algorithms with fine-grained granularity may be more parallelizable but may also incur higher overhead, while algorithms with coarse-grained granularity may be less parallelizable but may also incur lower overhead.

The optimal granularity for a given algorithm will depend on the specific characteristics of the problem, the available hardware and software resources, and the desired performance goals. Careful consideration of granularity is therefore an important aspect of the design of parallel algorithms.

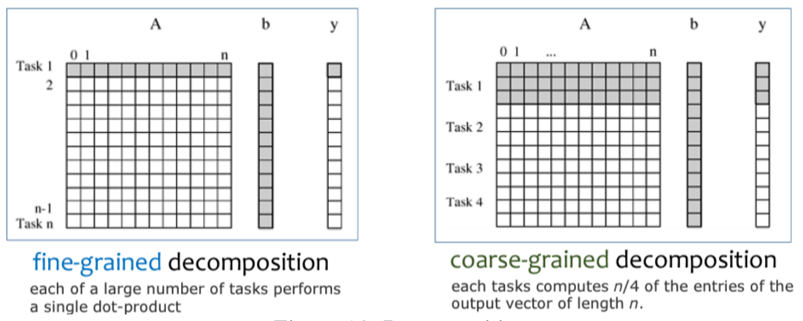

A
b
y
0 1
n
Task 1
2
n-1
Task n
fine-grained decomposition
each of a large number of tasks performs a single dot-product
A
b
y
0 1 ...
n
Task 1
Task 2
Task 3
Task 4
coarse-grained decomposition
each tasks computes n/4 of the entries of the output vector of length n.

Figure 16: Decomposition

5.4. Concurrency

Concurrency refers to the ability of a system to perform multiple tasks or processes at the same time. In the design of parallel algorithms, concurrency is an important consideration, as it allows the algorithm to take advantage of multiple processors or machines to perform the computation more efficiently.

By introducing concurrency into a parallel algorithm, it is possible to take advantage of the available hardware and software resources to perform the computation more efficiently and effectively. Careful consideration of concurrency is therefore an important aspect of the design of parallel algorithms.

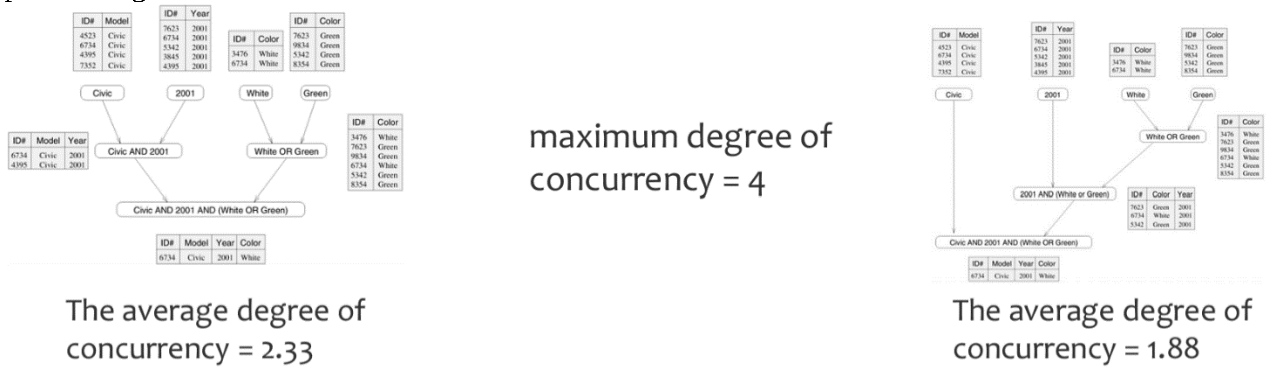

Civic
2001
White
Green
Civic AND 2001
White OR Green
Civic AND 2001 AND (White OR Green)
maximum degree of concurrency = 4
The average degree of concurrency = 2.33
2001 AND (White or Green)
The average degree of concurrency = 1.88

Figure 17: Concurrency

A concept related to granularity is that of degree of concurrency. The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its maximum degree of concurrency. In most cases, the maximum degree of concurrency is less than the total number of tasks due to dependencies among the tasks. The average degree of concurrency is the average number of tasks that can run concurrently over the entire duration of execution of the program, which is more useful indicator of a parallel program's performance.

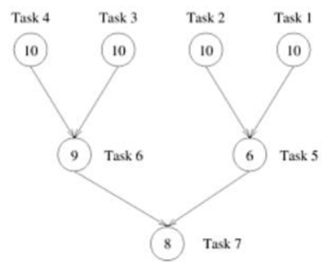
5.5. Critical Path

The critical path in a parallel algorithm is the sequence of tasks or operations that determines the minimum completion time for the algorithm. It is the longest path through the task graph and represents the tasks that cannot be started until all the preceding tasks have been completed.

In the design of parallel algorithms, the critical path is an important consideration, as it determines the bottleneck of the computation and the maximum speed at which the algorithm can be executed. By identifying the critical path and optimizing its performance, it is possible to improve the overall efficiency and speed of the algorithm.

A feature of a task-dependency graph that determines the average degree of concurrency for a given granularity is its critical path. In a task-dependency graph:

- Start nodes refer to the nodes with no incoming edges.
- Finish nodes refer to the nodes with no outgoing edges by.
- Critical path is known as the longest directed path between any pair of start and finish.
- Critical path length is the sum of the weights of nodes along this path, where the weight of a node is the size, or the amount of work associated with the corresponding task.
- The ratio of the total amount of work to the critical-path length is the average degree of concurrency. Therefore, a shorter critical path favors a higher degree of concurrency.



- the critical path length = 27
- the total amount of work = 63
- the average degree of concurrency = 2.33

Figure 18: Critical Path

By optimizing the critical path in a parallel algorithm, it is possible to improve the overall efficiency and speed of the computation. This is an important consideration in the design of parallel algorithms, as the critical path determines the maximum speed at which the algorithm can be executed.

5.6. Task Interaction

Task interaction refers to the communication and synchronization between tasks in a parallel algorithm. In the design of parallel algorithms, task interaction is an important consideration, as it can have a significant impact on the efficiency and performance of the algorithm.

The design of a parallel algorithm must consider the task interaction that is required to complete the computation and must ensure that the communication and synchronization between tasks is efficient and effective. Careful consideration of task interaction is therefore an important aspect of the design of parallel algorithms, as it can have a significant impact on the efficiency and performance of the algorithm.

A task interaction graph is a graphical representation of the communication and synchronization between tasks in a parallel algorithm. It is commonly used to model and analyze the interactions between different tasks in a parallel system, and to understand how changes in one task can affect the others.

In a task interaction graph, the nodes represent the tasks, and the edges represent the interactions between them. An interaction is a relationship in which one task depends on, communicates with, or synchronizes with another task to be completed.

Task interaction graphs can be used to model a wide range of parallel systems, including software systems, manufacturing systems, and scientific simulations. They are particularly useful for understanding and analyzing complex systems, as they provide a clear and visual representation of the interactions between different tasks and can help to identify potential issues or bottlenecks in the system.

There are several types of task interaction graphs, including directed graphs, in which the edges have a direction and represent a one-way interaction (e.g., task A depends on task B, but task B does not depend on task A), and undirected graphs, in which the edges do not have a direction and represent a mutual interaction (e.g., task A depends on task B, and task B depends on task A).

Task interaction graphs can be used to identify and analyze critical paths, identify bottlenecks and potential areas of conflict, and optimize the allocation of resources in a system. They are also useful for identifying and resolving interactions that may be causing delays or issues in a system.

Example: Sparse matrix-vector multiplication

Sparse matrix-vector multiplication is a common operation in scientific and engineering applications and can be implemented as a parallel algorithm to improve the efficiency and performance of the computation. To compute the product $y = Ab$ of a sparse $n\ x\ n$ matrix A with a dense $n\ x$ 1 vector b, you can use the following steps:

1. Divide the matrix A into p blocks, where p is the number of processors or machines available for the computation. Each block should contain a portion of the rows of the matrix.
2. Divide the vector b into p blocks, where each block contains a portion of the elements of the vector.
   The task interaction graph for this implementation is shown below:
   $[Matrix\ block\ 1]\ ---> \ [Vector\ block\ 1]\ ---> \ [Result\ block\ 1]$
   $[Matrix\ block\ 2]\ ---> \ [Vector\ block\ 2]\ ---> \ [Result\ block\ 2]$
   $\ldots$
   $[Matrix\ block\ p]\ ---> \ [Vector\ block\ p]\ ---> \ [Result\ block\ p]$
3. Assign each matrix block and its corresponding vector block to a separate processor or machine for concurrent processing.
4. On each processor or machine, perform the matrix-vector multiplication for the assigned block of the matrix and vector, using the formula:
   $$y[i] = sum(A[i][j] * b[j])$$
   Where i is the index of the row in the matrix, j is the index of the column in the matrix, and y[i] is the i-th element of the result vector y.
5. After all the blocks have been processed, combine the partial results to obtain the result vector y.

This approach allows the matrix-vector multiplication to be performed in parallel, with each processor or machine working on a smaller portion of the computation. This can greatly increase the speed and efficiency of the overall computation, particularly for large matrices and vectors.

It is important to note that this approach assumes that the matrix A is sparse, which means that it has many zero elements. If the matrix is dense, with a smaller number of zero elements, a different approach may be more efficient.

5.7. Processes and Mapping

In the design of parallel algorithms, processes and mapping refer to the assignment of tasks or units of work to processors or machines for concurrent execution. Processes are the units of work that are executed concurrently in a parallel algorithm. They can be individual tasks, such as matrix-vector multiplications or data filtering operations, or they can be larger units of work, such as subproblems or subdomains.

Mapping is the process of assigning processes to processors or machines for concurrent execution. The goal of mapping is to optimize the performance of the parallel algorithm by assigning the processes to the available resources in a way that minimizes communication overhead and maximizes load balancing.

By carefully designing the processes and mapping in a parallel algorithm, it is possible to optimize the performance and efficiency of the computation. This is an important consideration in the design of parallel algorithms, as the assignment of processes to processors or machines can have a significant impact on the overall performance of the algorithm.

CHAPTER 6: PARALLEL ALGORITHM IMPLEMENTATION

After we know about parallel algorithms, we will look about how parallel algorithms are implemented in real life cases. Parallel algorithms can be implemented using a variety of programming languages and tools, including MATLAB and Python.

MATLAB is a commercial software platform that is widely used for scientific and engineering computations and has built-in support for parallel computing using its Parallel Computing Toolbox.

Python is a general-purpose programming language that has a large and active community of users, and it has several libraries and frameworks that support the implementation of parallel algorithms, such as the multiprocessing and concurrent.

Both Matlab and Python offer a wide range of tools and libraries for implementing and testing parallel algorithms, including support for parallel programming models such as data parallelism, task parallelism, and pipeline parallelism, as well as tools for analyzing and optimizing the performance of parallel algorithms.

6.1. Parallel MATLAB

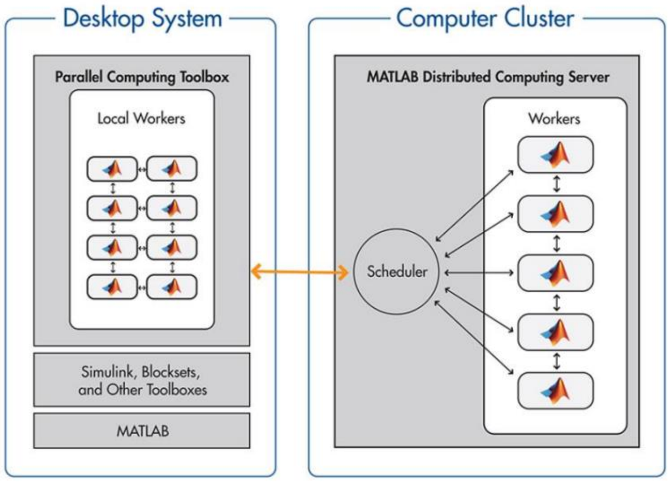

Desktop System
Parallel Computing Toolbox
Local Workers
Simulink, Blocksets, and Other Toolboxes
MATLAB
Computer Cluster
MATLAB Distributed Computing Server
Workers
Scheduler

Figure 19: Parallelization in MATLAB

There are two ways to write a parallel MATLAB Program:

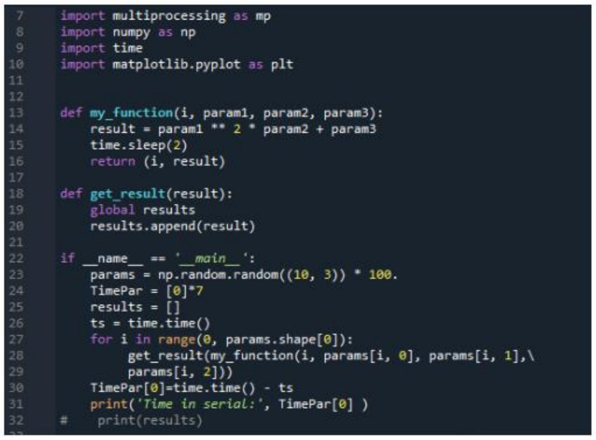
1. **parfor**, the simplest path to parallelism is the parfor statement, which indicates that a given for loop can be executed in parallel. When the "client" MATLAB reaches such a loop, the iterations of the loop are automatically divided up among the workers, and the results gathered back onto the client. Using parfor requires that the iterations are completely independent; there are also some restrictions on data access.
    - **Requirements for parfor loops**
        - Task independent
        - Order independent
    - **Constraints on the loop body**
        - Cannot "introduce" variables (e.g., eval, load, global, etc.)
        - Cannot contain break or return statements
        - Cannot contain another parfor loop
2. **spmd,** stands for single program multiple data. The spmd statement lets you define a block of code to run simultaneously on multiple workers.
    - MATLAB sets up one special worker called the client.

- MATLAB sets up the requested number of workers, each with a copy of the program. Each worker “knows" it's a worker, and has access to two special functions: numlabs(), the number of workers; labindex(), a unique identifier between 1 and numlabs(). (The empty parentheses are usually dropped, but remember, these are functions, not variables!)
- If the client calls these functions, they both return the value 1! That's because when the client is running, the workers are not.
- The client could determine the number of workers available by n = matlabpool ('size')

SPMD means that the identical code runs on multiple workers. You run one program in the MATLAB client, and those parts of it labelled as spmd blocks run on the workers. When the spmd block is complete, your program continues running in the client. The “multiple data” aspect means that even though the spmd statement runs identical code on all workers, each worker can have different, unique data for that code. So multiple data sets can be accommodated by multiple workers. Typical applications appropriate for spmd are those that require running simultaneous execution of a program on multiple data sets, when communication or synchronization is required between the workers.

6.2. Parallel Python

The most common way to code parallel program in Python is using the multiprocessing library. By using the standard multiprocessing module, we can efficiently parallelize simple tasks by creating child processes. This module provides an easy-to-use interface and contains a set of utilities to handle task submission and synchronization.

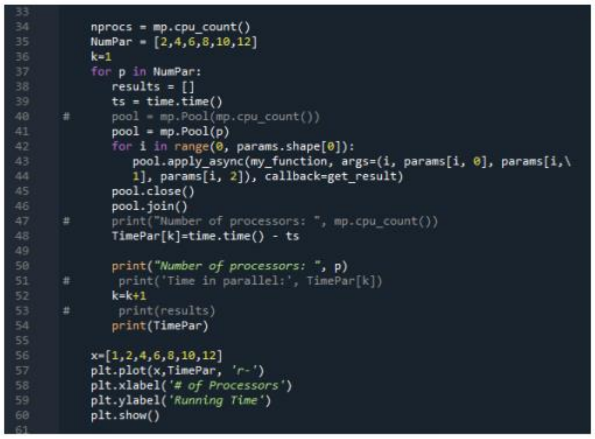


```python
import multiprocessing as mp
import numpy as np
import time
import matplotlib.pyplot as plt


def my_function(i, param1, param2, param3):
    result = param1 ** 2 * param2 + param3
    time.sleep(2)
    return (i, result)

def get_result(result):
    global results
    results.append(result)

if __name__ == '__main__':
    params = np.random.random((10, 3)) * 100.
    TimePar = [0]*7
    results = []
    ts = time.time()
    for i in range(0, params.shape[0]):
         get_result(my_function(i, params[i, 0], params[i, 1],\
         params[i, 2]))
    TimePar[0]=time.time() - ts
    print('Time in serial:', TimePar[0] )
#     print(results)
```



```python

    nprocs = mp.cpu_count()
    NumPar = [2,4,6,8,10,12]
    k=1
    for p in NumPar:
       results = []
       ts = time.time()
#      pool = mp.Pool(mp.cpu_count())
       pool = mp.Pool(p)
       for i in range(0, params.shape[0]):
          pool.apply_async(my_function, args=(i, params[i, 0], params[i,\
          1], params[i, 2]), callback=get_result)
       pool.close()
       pool.join()
#      print("Number of processors: ", mp.cpu_count())
       TimePar[k]=time.time() - ts

       print("Number of processors: ", p)
#       print('Time in parallel:', TimePar[k])
       k=k+1
#       print(results)
       print(TimePar)

    x=[1,2,4,6,8,10,12]
    plt.plot(x,TimePar, 'r-')
    plt.xlabel('# of Processors')
    plt.ylabel('Running Time')
    plt.show()
```

Figure 20: Code Example for Parallelization in Python

Another way to code parallel program in Python is using the IPython Parallel Framework. IPython parallel package provides a framework to set up and execute a task on single, multi-core machines and multiple nodes connected to a network. In IPython.parallel, you must start a set of workers called Engines which are managed by the Controller. A controller is an entity that helps in communication between the client and engine. In this approach, the worker processes are started separately, and they will wait for the commands from the client indefinitely.

References:

[1] "Introduction to Parallel Algorithms" by J. H. Reif (Cambridge University Press, 1998)
[2] "Parallel Algorithms" by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani (McGraw-Hill, 2006)
[3] "The Theory of Parallel Processes" by G. L. Chaitin and J. K. Mertzios (Cambridge University Press, 2015)
[4] "Introduction to Algorithms" by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein (MIT Press, 2009)
[5] "Data Structures and Algorithms" by A. V. Aho, J. E. Hopcroft, and J. D. Ullman (Addison-Wesley, 1983)
[6] "Parallel Computing with MATLAB" by K. K. Puri, M. A. Sabin, and G. R. Dattatreya (CRC Press, 2018)
[7] "Parallel and Concurrent Programming in Python" by E. L. Rios and M. O. Faruque Sarker (O'Reilly, 2019)
[8] "Parallel Computing: Theory and Practice" by Michel J. Quinn (4th edition) - 2020
[9] "Performance Measurement and Analysis" by George Karypis - 2002
[10] "Parallel Algorithms and Applications" by Todd C. Neff - 2000
[11] "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar (2003)
[12] "OpenMP: A User's Guide" by Barbara Chapman, Gabriele Jost, and Ruud van der Pas (2013)
[13] "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser (2013)
[14] "Parallel Computer Architecture: A Hardware/Software Approach" by David E. Culler, Jaswinder Pal Singh, and Anoop Gupta (1999)
[15] "Parallel Algorithms" by Todd C. Neller (2005)
[16] “Parallel Processing in Python” by SonuGeorge in GeeksforGeeks (2019)