AA LAB ASSIGNMENT | PARALLEL ALGORITHM

TITLE: Analysis, Proof of Analysis and Implementation of **Parallel Algorithm**

Introduction

A parallel algorithm is a type of algorithm that allows multiple processors or threads to work together simultaneously to solve a problem, in order to reduce the overall time required to solve the problem. This approach is especially useful for solving large-scale problems that require a lot of computational power. To design a parallel algorithm, one needs to consider factors such as the number of processors available and the complexity of the problem being solved. Parallel algorithms come in different types, depending on the hardware and problem at hand.

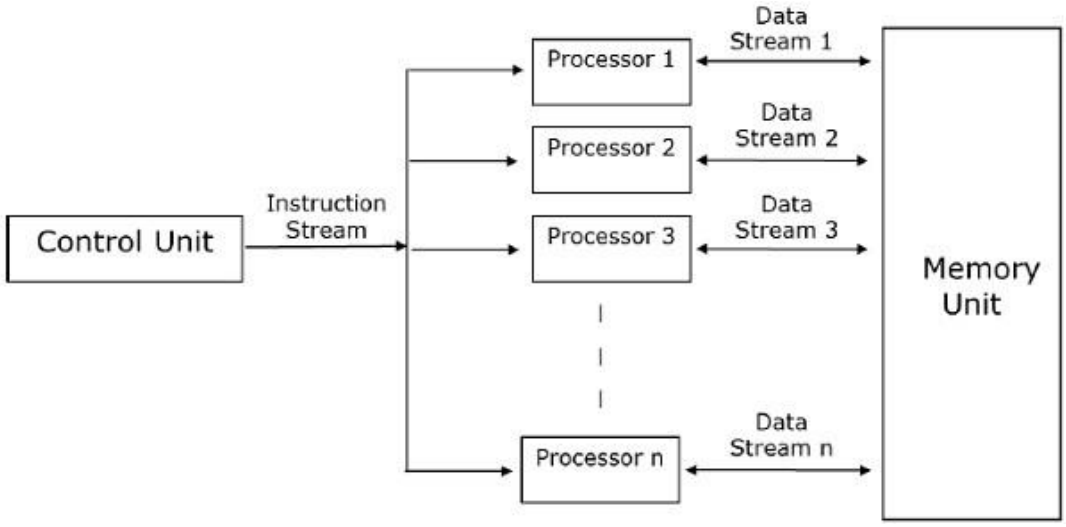

Figure 1: Model of Parallel Algorithm

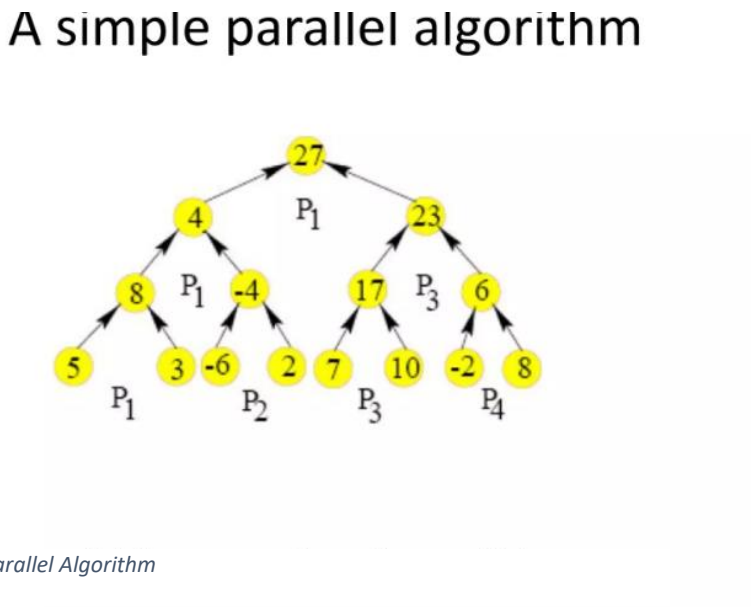

Figure 2: Example of Parallel Algorithm

IMPLEMENTATION

```
#include <iostream>
#include <omp.h>

int main() {
   int n = 100;
   int* arr = new int[n];
   int sum = 0;

   // Initialize array
   for (int i = 0; i < n; i++) {
      arr[i] = i + 1;
   }

   // Calculate sum in parallel using OpenMP
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++) {
      sum += arr[i];
   }

   // Output the sum
   std::cout << "Sum: " << sum << std::endl;

   // Free memory
   delete[] arr;

   return 0;
}
```

Output

```
Sum: 5050
```

Time Complexity

The time complexity of the parallel summation algorithm depends on the number of processors or threads available and the size of the input array.

Assuming that we have p processors or threads available and an input array of size n, the time complexity of the algorithm can be expressed as O(n/p + log p), where n/p represents the time required to compute the partial sums on each processor or thread, and log p represents the time required to combine the partial sums using a binary tree reduction.

In the case of the example code provided earlier, we have used OpenMP to parallelize the summation of an array. OpenMP uses a shared-memory model, which means that all processors or threads have access to a shared memory space. The time complexity of this approach is similar to the one described above, with the number of processors or threads representing the degree of parallelism.

ADVANTAGES

- **Reduced execution time:** Parallel algorithms can divide a problem into smaller parts and solve them simultaneously, which can lead to faster execution times compared to sequential algorithms.
- **Increased processing power:** By utilizing multiple processors or threads, parallel algorithms can harness the power of multiple CPUs or cores, allowing for increased processing power and improved performance.
- **Scalability:** Parallel algorithms can be scaled to work with larger input sizes and accommodate more processors or threads, making them well-suited for handling big data applications.
- **Efficiency:** Parallel algorithms can reduce the amount of energy used by individual processors, making them more energy-efficient than sequential algorithms.

DISADVANTAGES

- **Complexity:** Parallel algorithms can be complex and difficult to design and implement, requiring specialized knowledge of parallel programming paradigms and architectures.
- **Synchronization and communication overhead**: In order to ensure correct results, parallel algorithms often require synchronization and communication between processors or threads, which can introduce overhead and increase the complexity of the algorithm.
- **Limited parallelism:** Not all algorithms can be parallelized effectively, as some problems may not lend themselves well to parallelization or may have limited opportunities for parallelism.
- **Cost:** Parallel computing requires hardware and software that can support parallelism, which can be expensive to acquire and maintain. Additionally, the

development of parallel algorithms may require specialized skills and resources, which can also increase costs.

REAL LIFE APPLICATIONS:

Weather forecasting: Weather forecasting requires simulating complex systems of equations that model atmospheric conditions. Parallel algorithms can be used to speed up the simulations and improve the accuracy of the forecasts.

Medical imaging: Medical imaging techniques like computed tomography (CT) and magnetic resonance imaging (MRI) produce large amounts of data that need to be processed in real-time. Parallel algorithms can be used to accelerate image processing and improve the accuracy of diagnoses.

Financial modelling: Financial models used in areas such as risk analysis, portfolio management, and trading require the processing of large amounts of data. Parallel algorithms can be used to speed up the calculations and improve the accuracy of predictions.

Video rendering: Creating high-quality animations or special effects in movies and video games requires the rendering of complex 3D models. Parallel algorithms can be used to speed up the rendering process and reduce the time needed to produce the final product.

Genome sequencing: Sequencing the human genome involves analysing huge amounts of genetic data. Parallel algorithms can be used to accelerate the analysis and improve our understanding of genetic disorders and diseases.

OPTIMIZATIONS AND ADVANCEMENTS:

Hybrid algorithms: Hybrid algorithms combine multiple parallel paradigms, such as shared memory and distributed memory, to improve performance and scalability.

GPU computing: Graphics Processing Units (GPUs) are increasingly being used to accelerate parallel computations due to their high memory bandwidth and processing power.

Communication optimization: Communication between processors or threads can be a bottleneck in parallel algorithms, so optimizing the communication patterns and reducing overhead can significantly improve performance.

Load balancing: Uneven distribution of workloads among processors can result in idle time and reduced efficiency, so load balancing techniques aim to distribute the workload evenly among processors.

Task-based parallelism: Task-based parallelism allows for more dynamic parallelism by breaking down a problem into smaller tasks that can be executed independently and concurrently.

Auto-tuning: Auto-tuning techniques automatically optimize parameters such as data layout, parallelization strategy, and communication patterns based on the input data and the available hardware, improving performance without requiring manual tuning.