Project Report

Name: Nazifa Mosharrat

Matricula: 682012

Department: Computer Science –AI Track

Course: SPM

Introduction

This project implements a distributed wavefront computation on an N X N matrix, where each diagonal element is calculated using a dot product between vectors derived from the matrix. In this project to parallel versions are developed, one using Fast-Flow and another one is MPI for distributed parallelism across a cluster of machines. The goal is to accelerate the computation by leveraging both shared-memory and distributed-memory parallelism, comparing the efficiency and scalability of the two approaches.

Sequential Wavefront Computation

The wavefront computation involves filling a matrix with computed values based on a specific pattern. The matrix is divided into diagonals, and each diagonal's elements depend on values from previous diagonals, introducing data dependencies that influence how the computation is parallelized. The sequential wavefront computation has been calculated by following process

- The matrix is processed in a diagonal wavefront manner. For each upper diagonal, elements are computed using a dot product of vectors from the matrix, followed by taking the cubic root of the result.
- The sequential algorithm processes one diagonal at a time, making it inherently limited by the sequential nature of these dependencies.

Parallelization Via Fast-Flow

Fast-Flow is a parallel programming framework that abstracts the complexity of low-level threading. In this project the strategy for parallelization involves using Fast-Flow's *"ParallelFor"* construct to parallelize the computations across different threads.

The steps for parallelization the wavefront computation-

- Matrix initialized with default values which ensuring that the diagonal elements were set to a normalized value.
- The outer loop over the diagonals was retained, but the inner loop that iterated over the elements of the diagonal was parallelized using Fast-Flow's *"ParallelFor"*. Each thread independently computed elements of the wavefront, reducing synchronization overhead.
- Here synchronization is required to respect the dependencies between diagonals. Here Fast-Flow manages synchronization internally, ensuring that all threads have consistent views of the shared matrix data without explicit locking mechanisms, reducing overhead and improving performance.

Performance Analysis:

The performance is assessed in terms of execution time, speedup, and efficiency. The program is tested on N X N matrix. $N \epsilon [70,100,500,700]$. The execution times for the sequential and parallel implementations:

Matrix Size (N)	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency (%)
70	0.002857	0.110527	0.0259	2.59%
100	0.007394	0.111499	0.0663	6.63%
500	0.694996	0.189602	3.664	366.40%
700	1.86083	0.304212	6.114	611.40%

Table 1: Performance analysis table for Fast-Flow

On **[Table 1]** the speedup is calculated as $speedup = \frac{T_{Sequential}}{T_{Parallel}}$.

Here the task will be executed across the core internally based on the available core without explicitly defining it. Since the exact number of cores efficiency is expressed relative to the speedup percentage.

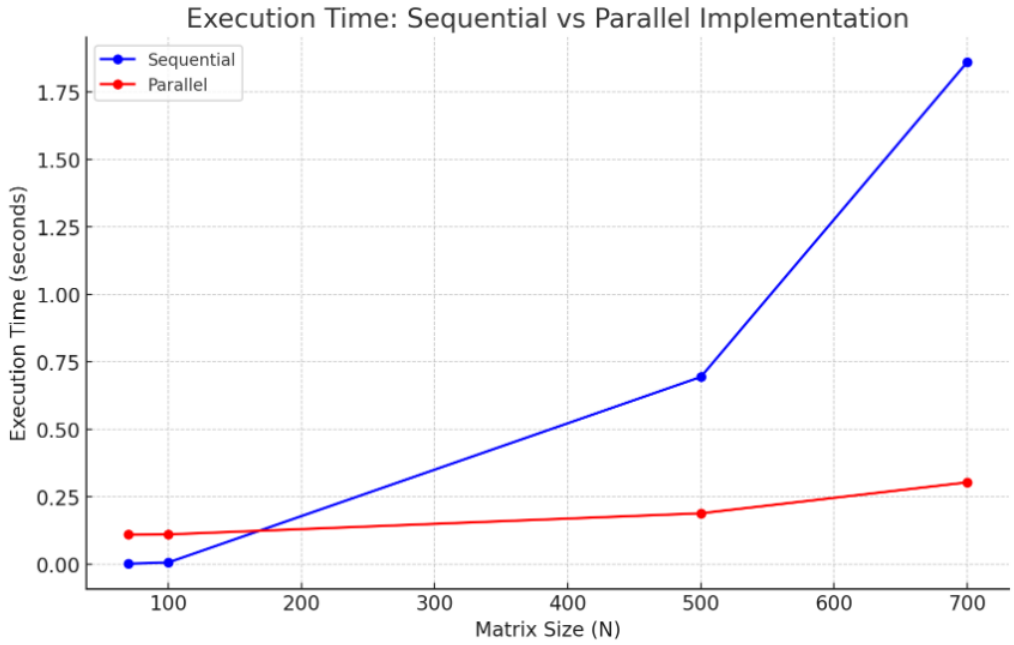

Chart 1: Performance analysis chart for Fast-Flow

From **[Chart 1]** plot efficiency is very low for small matrices, where the parallel overhead dominates. Here matrix size increases, efficiency improves, reflecting better utilization of the parallel resources.

The parallel implementation introduces overhead, which is not amortized for small matrix sizes. This explains the poor performance for small matrices. For small matrices, the overhead of

managing parallel threads and synchronizing them outweighs the benefits of parallel computation, leading to worse performance compared to the sequential approach. The benefits of parallel computation become evident as the problem size increases, showing that parallelism is more effective for larger matrix sizes.

Parallelization Via MPI

In this section the implementation is carried out using the MPI library. The code is structured to execute the parallel computation, with the following key components:

- Generates random values to initialize the matrix elements.
- Calculates the dot product of two vectors and applies a cubic root to the result.
- Separate functions are implemented for sequential and parallel wavefront computation.
- Here we compare the output of the sequential and parallel algorithms to ensure correctness.
- Here we measure the execution time for both sequential and parallel computations.

The steps for parallelization the wavefront computation-

- Initializing the MPI environment, sets up the matrix, and manages the timing and synchronization of the parallel processes.
  - ***MPI_Init*** -Initializes the MPI environment.
  - ***MPI_Comm_rank*** -Retrieves the rank of the process which is the unique ID of each process.
  - ***MPI_Comm_size-*** Gets the total number of processes in the communicator.

- The "*random()" function* is same as sequential version which generates random integers within a specified range [min, max]. The generator is static to ensure consistent random numbers across different calls within the same process.
- The *"dot_product()" function is* computing the dot product of two vectors and then returns the cubic root of the result. It takes 2 vector v1 and v2 as parameters of the same size whose dot product is to be computed.
- Here the main "*parallel_wavefront_computation()*" function is responsible for distributing the wavefront computation across multiple processes using MPI, especially message passing techniques. This function will take matrix need computation, the size of the matrix, the rank of the current process and finally the total number of processes.

The process of parallel computation will be done by following the below steps-

- Firstly, The matrix rows will be divided among the processes, with each process handling a subset of rows. The computation is spread across processes by computing the elements of the matrix in a wavefront pattern. Each process is assigned a block of rows to compute based on its rank. Before computing each diagonal, processes are synchronized using `MPI_Barrier` to ensure they all start computing the same diagonal simultaneously.
- Then each process computes the required elements of the current diagonal for its assigned rows. The results are shared between neighboring processes using `MPI_Send` and `MPI_Recv` to ensure the necessary data is available for the next diagonal computation.
- Finally, another `MPI_Barrier` is used after computing each diagonal to ensure all processes have completed their work before moving on.
- Here to verify the output of the parallal version is consistence with the sequential version a function *"compare_matrices()"* has been used to compare the result of wavefront computation of the sequential and parallel is same or not.

Performance Analysis:

The experiments were conducted on a shared computing environment, with different matrix sizes (N = 500 and N = 1500) and varying numbers of processors (np = 4 and np = 8). The results are summarized below:

Matrix Size (N)	Node	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency
500	4	0.063625	0.036529	1.74	0.435
1500	4	1.490898	0.858430	1.74	0.435
1500	8	1.493784	0.501679	2.98	0.372

Table 2: Performance analysis table for MPI

In **[Table 2]**, the Speedup is calculated as $speedup = \frac{T_{Sequential}}{T_{Parallel}}$.

In addition to that the efficiency is calculated as $efficiency = \frac{Speedup}{Number\ of\ process}$.

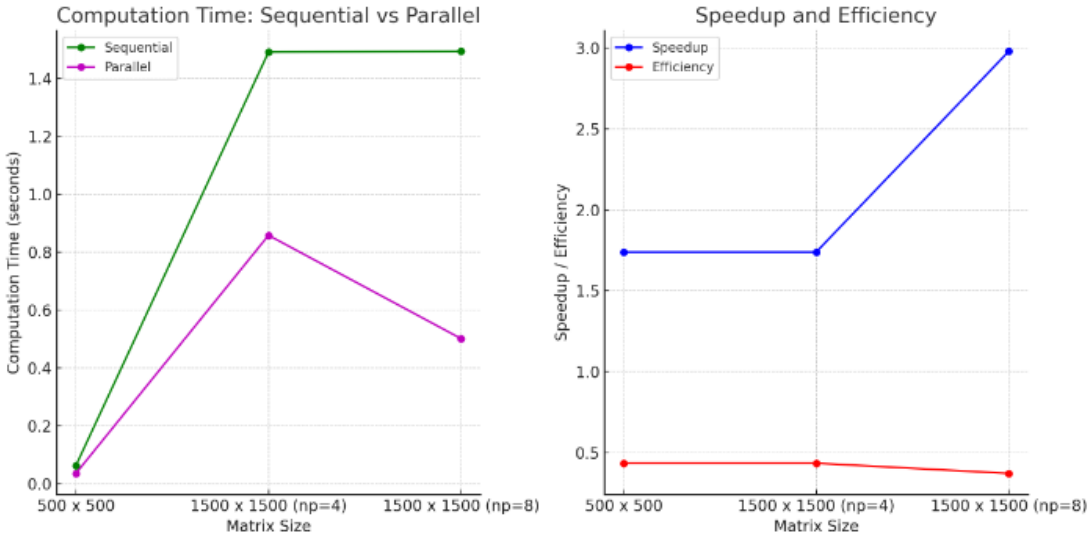

Chart 2: Performance analysis chart for PMI

From **[Chart 2]** we can compare the computation time for the sequential and parallel versions across different matrix sizes. The parallel version consistently outperforms the sequential version, as indicated by the lower computation times. The speedup increases as the number of processors increases, demonstrating the benefits of parallelization.

On the other hand, the efficiency decreases slightly as the number of processors increases, indicating that although the computation time for the sequential and parallel versions across different matrix sizes. This is common in parallel computing due to communication overhead and synchronization costs. In a parallel system, processors often need to communicate with each other, especially when sharing or exchanging data. As the number of processors increases, the communication overhead also increases, which can reduce overall efficiency. Here in the MPI par parallel system, the message-passing algorithms require synchronization points where all processors must wait for each other to reach a certain point in the computation. This can cause some processors to idle while waiting for others to catch up, reducing the system's efficiency.

Compilation and Execution:

To efficiently compile and execute both MPI and Fast-Flow-based implementations of the wavefront computation, we utilized a Slurm script. The script uses Slurm directives to manage output:

- #SBATCH --output=wavefront_output.txt: Specifies the name of the output file.
- #SBATCH --error=wavefront_error.txt: Specifies the name of the error file.

This ensures that the output and error messages for submitted jobs are stored separately, for further debugging and analysis.

Conclusion

From the above experiment and analysis, we can say that the MPI and Fast-Flow both provide effective for accelerating wavefront matrix computations, with each framework offering distinct advantages depending on the problem size and the available computational resources. MPI excels in scalability and fine-grained control over parallel tasks, making it suitable for large-scale computations but the efficiency gets reduced for the small-scale computation because of resource overhead problems. Fast-Flow, however, offers ease of use and efficient parallelization for moderately large matrices. Both methods consistently yield correct results.