SPM Project - Wavefront

Davide Bulotta - 596782

October 2024

1 Design Choices

This project aimed to address the wavefront problem through multiple approaches, each aiming to leverage different computational paradigms and optimizations. In line with the project requirements, I implemented solutions using FastFlow and MPI, ensuring that various strategies were explored and evaluated.

The following solutions were developed:

1.1 Sequential Version

- **A standard sequential version**: This version was designed as a baseline for comparison.
- **A sequential version employing a cache-oblivious algorithm**: The aim was to improve memory access on the cache.
- **Two versions using cache-oblivious algorithms and AVX instructions**: The first version utilized 64-bit values, while the second operated with 32-bit values, allowing for an evaluation of performance differences based on data precision and speed.

1.2 FastFlow-based Solutions

- **FastFlow's `parallel_for` construct**: This version was designed to parallelize the computation while maintaining simplicity.
- **FastFlow's `parallel_for` with the cache-oblivious algorithm**: This approach aimed to optimize memory usage alongside parallel execution.
- **FastFlow's farm construct with the cache-oblivious algorithm**: The objective was to explore the benefits of task farming in a more complex parallel framework.

1.3 Distributed Version (MPI)

- **A distributed version implemented using MPI**: This version was designed for distributed execution and incorporated the cache-oblivious algorithm to optimize memory access (were possible).

Each of these approaches was carefully chosen to evaluate the trade-offs between parallelism, memory access optimization, and computational efficiency, particularly focusing on the performance gains when using cache-oblivious algorithms and AVX instructions in different computational environments.

1.4 Compiler Optimization Parameters

To enhance performance, the following compiler flags were selected after several tests:

- `-march=native`: Optimizes code for the CPU, leveraging all available hardware features.
- `-ffast-math`: Allows aggressive floating-point optimizations, improving numerical performance with a minor trade-off in precision.
- `-O3`: Enables high-level optimizations such as vectorization, inlining, and loop unrolling for maximum performance.

These flags ensure the code is optimized for the specific processor, resulting in faster execution through specialized hardware and advanced compiler optimizations.

2 Sequential Approach

The first approach for the sequential problem was a brute-force solution without any focus on efficiency. This was done intentionally to better understand the magnitude and nature of the problem. Afterward, the implementation of the cache-oblivious algorithm, allowed to compare the performance gain between the simple sequential implementation and the optimized cache-oblivious approach. The performance increase was significant, especially for larger matrices.

For $N < 2000$, the performance difference between the two approaches is not very noticeable. However, as the matrix size increases, the difference becomes much more apparent. The performance improvement becomes drastic as N grows beyond 2000.

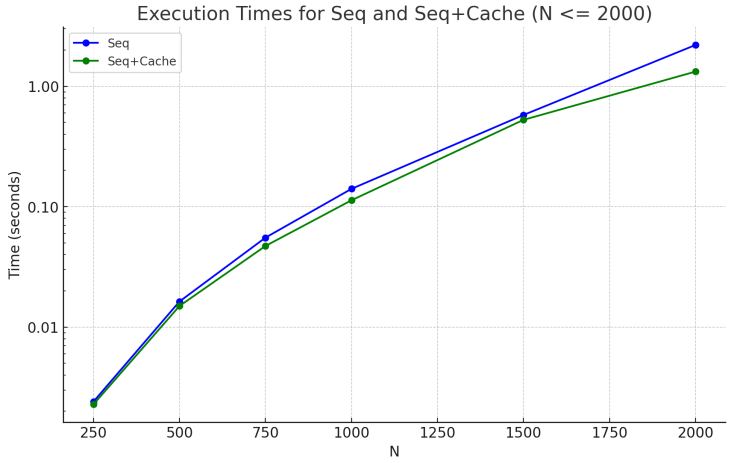

Figure 1: Performance comparison for $N \leq 2000$

For larger matrix sizes, such as $N = 10000$, the performance difference is substantial. The performance gain between the brute-force sequential approach and the cache-oblivious approach for $N = 10000$ is calculated as follows:

$$\frac{\text{Seq Time}}{\text{Seq+Cache Time}} = \frac{1476.29 \text{ seconds}}{167.191 \text{ seconds}} \approx 8.83$$

This means the cache-oblivious version is approximately 8.83 times faster than the brute-force implementation for $N = 10000$.

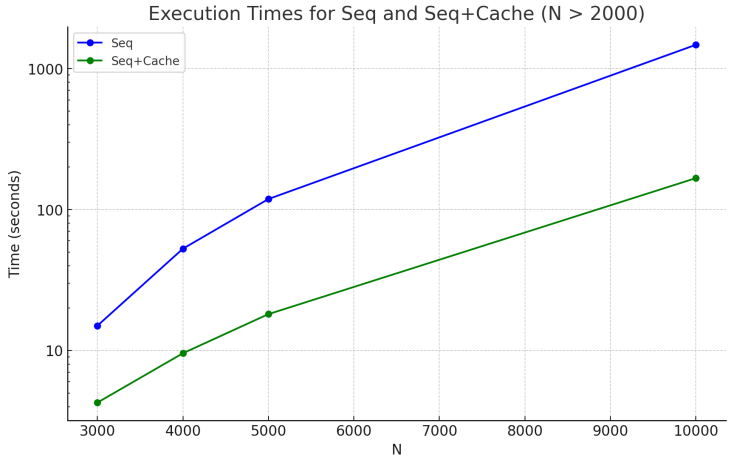

Figure 2: Performance comparison for $N > 2000$

3 Sequential AVX Implementation

In an attempt to further improve the sequential version, using the SIMD approach (Single Instruction - Multiple Data): The AVX (Advanced Vector Extensions) instructions with 256-bit registers, allowed to process four 64-bit double values togeher. This implementation, like the previous one, includes the cache-oblivious algorithm to maintain as much efficiency as possible. The performance improvement was noticeable, but was implemented an additional variant to test the potential of AVX instructions further.

Since the access to 512-bit registers is not possible (which can handle eight 64-bit doubles), Intentinally was tested the same problem with reduced precision by using 256-bit registers with 32-bit floats, thus handling eight 32-bit floats per register.

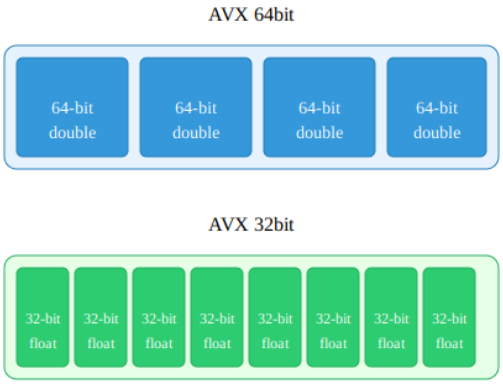

Figure 3: AVX64bit vs AVX32bit structure

In the graph [4], both AVX64bit and AVX32bit implementations show a logarithmic increase in execution time as N increases. The AVX32bit version consistently outperforms AVX64bit, achieving nearly twice the performance due to processing twice as many elements per register. For $N = 2000$, AVX64bit takes about 1 second, while AVX32bit completes in 0.62026 seconds.

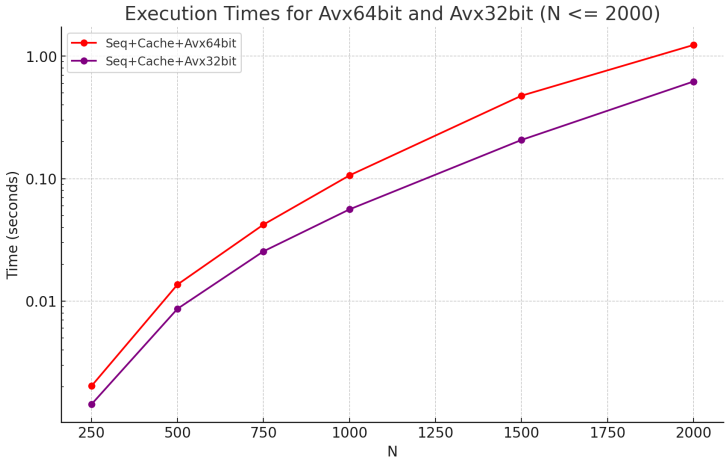

Figure 4: AVX64bit vs AVX32bit comparison for $N \leq 2000$

For larger matrix sizes, the performance gap widens significantly. At $N = 10000$, AVX64bit takes 127.677 seconds, while AVX32bit finishes in about 65.254 seconds. This means that the AVX32bit implementation is approximately

$$\frac{\text{AVX64bit (Time)}}{\text{AVX32bit (Time)}} = \frac{127.677 \text{ seconds}}{65.254 \text{ seconds}} \approx 1.96$$

1.96 times faster than the AVX64bit version. The ability to process more elements per register provides a clear advantage as N increases.

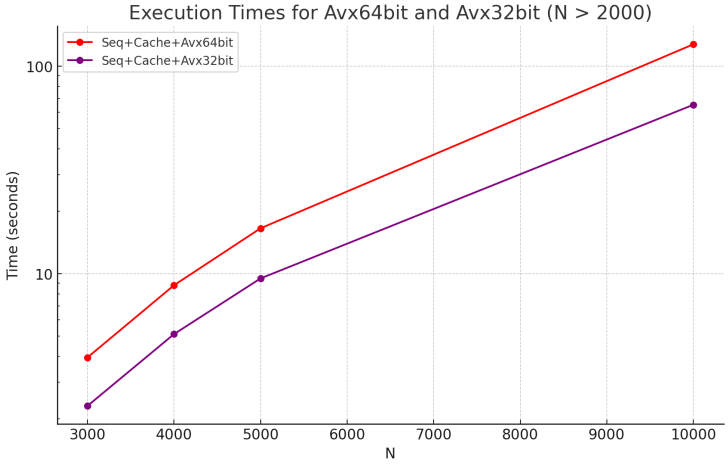

Figure 5: AVX64bit vs AVX32bit comparison for $N > 2000$

3.1 Precision Trade-offs

While the AVX32bit implementation offers significant performance improvements, it comes with a trade-off in precision. Using 32-bit floats instead of 64-bit doubles reduces the accuracy of calculations, which can be critical for some applications. The difference in precision is minimal for small matrix sizes, but it can grow as N increases.

For example, with $N = 1000$, the results for 64-bit and 32-bit implementations differ only in the last decimal places. However, at $N = 10000$, the difference becomes more noticeable, as shown in the following comparison:

N	**64-bit Result**	**32-bit Result**
1000	170.678669	170.678665
2000	338.981510	338.981537
4000	674.686285	674.686340
5000	842.325310	842.325275
10000	1679.552590	1679.552734

Table 1: Precision differences between 64-bit and 32-bit implementations

3.1.1 General error propagation

It is important to note that the last element in the matrix (the last element in the first row) is particularly susceptible to error propagation. If the least significant digits of all preceding values are incorrect, this accumulated error will cause the final element to be more imprecise.

A more general form to capture the error propagation of the last element in sequential computations is given by:

$$\epsilon_{total}(e_{1,N}) = \sum_{k=1}^{N-2} \epsilon(v_{row}^k \cdot v_{col}^k) + \epsilon(v_{row}^{N-1} \cdot v_{col}^{N-1})$$

where:

- $\sum_{k=1}^{N-2} \epsilon(v_{row}^k \cdot v_{col}^k)$ represents the propagated error from previous elements.
- $\epsilon(v_{row}^{N-1} \cdot v_{col}^{N-1})$ is the error introduced by the current operation (in this case the last one).

Thus, the total error at any element $e_{i,j}$ accumulates as:

$$\epsilon_{i,j} = \epsilon_{\text{propagated}} + \epsilon_{\text{new operation}}$$

This describes how errors propagate and accumulate through sequential computations, especially in matrix algorithms like wavefront computations.

4 Parallel Implementation

Initially, was implemented a brute-force parallel version using the `parallel_for` approach. This version was then compared to a more efficient implementation with the cache-oblivious algorithm. Was conducted extensive tests, varying the number of workers from 5 to 40 and using different matrix sizes (N) from 250 to 15,000. Below, the graph comparing the performance between the two versions with $N = 10000$ and the same number of workers.

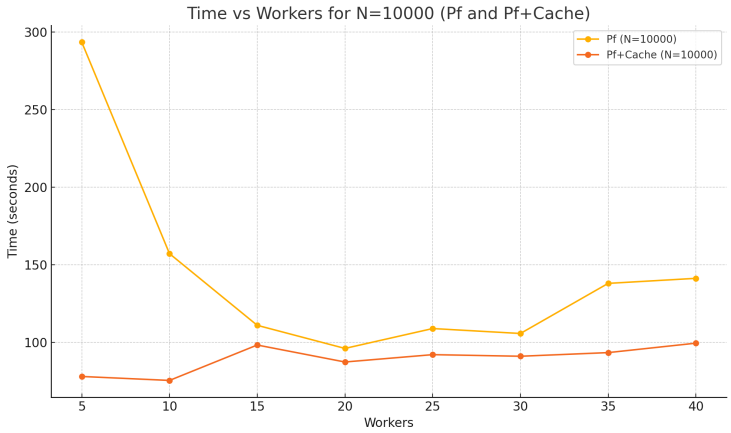

Figure 6: Performance comparison: brute-force parallel vs cache-oblivious algorithm $N = 10000$

4.1 Farm Architecture

A more complex solution was the implementation of a farm pattern to parallelize the task. The farm-based approach is a more structured method for distributing work efficiently across multiple workers.

4.1.1 Diagonal Emitter Description

The diagonal emitter is responsible for distributing tasks to the available workers (W). After all tasks are dispatched, the emitter waits, allowing all resources to be used by the worker threads. Once the wavefront computation is completed, the emitter sends an End-Of-Stream (EOS) signal to all workers, indicating that no more tasks are available.

4.1.2 Diagonal Worker Description

Each diagonal worker receives tasks from the emitter. Using the cache-oblivious algorithm, the worker computes the current matrix element and signals that the task has been completed. The worker continues to wait for tasks until the EOS signal is received from the emitter, indicating the completion of all tasks.

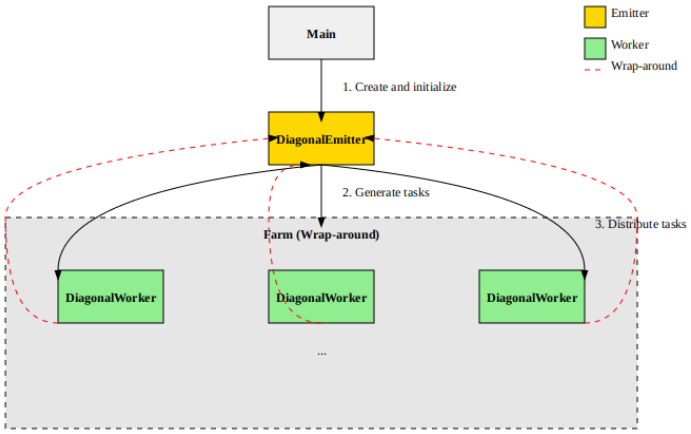

Figure 7: Structure of the Farm

4.1.3 Farm Performance Results

Different performance tests was conducted on the wavefront farm, focusing on varying matrix sizes and different numbers of workers. The purpose of these tests was to thoroughly evaluate how the system behaves under different computational loads and parallelization levels. In particular, the relationship between matrix size and the number of workers provides deep insight into the trade-offs between computational complexity and parallel efficiency.

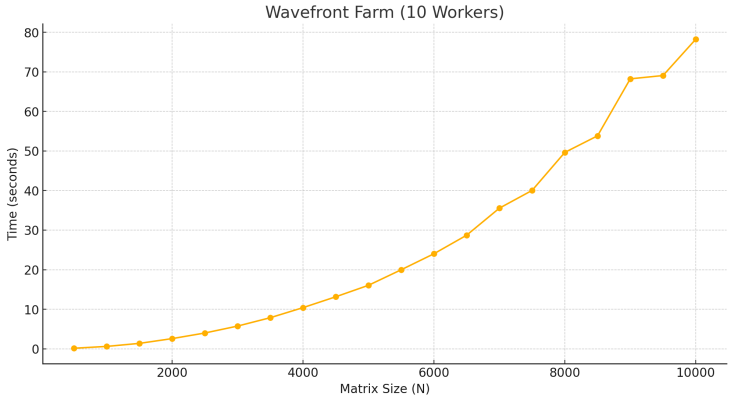

Figure 8: Performance of the Farm implementation

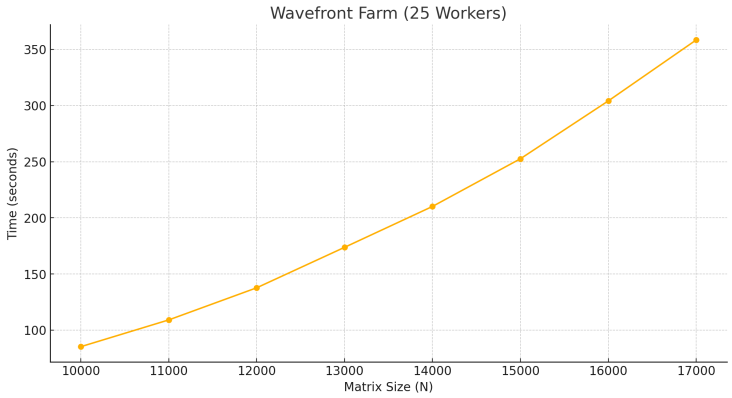

Figure 9: Performance of the Farm Implementation

Both graphs reveal crucial insights into the performance characteristics of the wavefront farm under different conditions. While parallelism improves performance to some extent, the benefits diminish as the number of workers increases and the matrix size grows. Synchronization and task allocation strategies likely contribute to the observed bottlenecks. Optimizing these aspects could lead to substantial improvements in handling larger matrices more efficiently across multiple workers.

4.1.4 Results and machine compare

Extensive tests was conducted on each parallel version individually. Using a fixed number of workers (20) to facilitate performance comparison across different matrix sizes. These tests were performed on multiple machines to account for variations we can obtain with varying architectures of hardware, particularly noticeable in the initial portions of the graphs, where results tend to differ depending on the hardware. The machines are three, and they have the following hardware specs:

- **Numa** with 2 socket AMD Epyc 7301 - 16 core - 32 threads - L1 (96KB per core) Total: 1536KB - L2 (512KB per core) Total: 8192 KB - L3 64MB (shared)
- **Intel machine** with 1 socket i7-13700k - 16 core (8P - 8E) - 24 threads - L1 (80KB per core P) (96KB per core E) Total: 640KB + 768KB = 1408KB - L2 (2MB per core) Total: 24MB - L3 30MB (shared)
- **Nvidia DGX** with 2 socket AMD EPYC 7742 - 64 core - 128 threads - L1 (96KB per core) Total: 6144KB - L2 (512KB per core) Total: 32768KB - L3 256MB (shared)

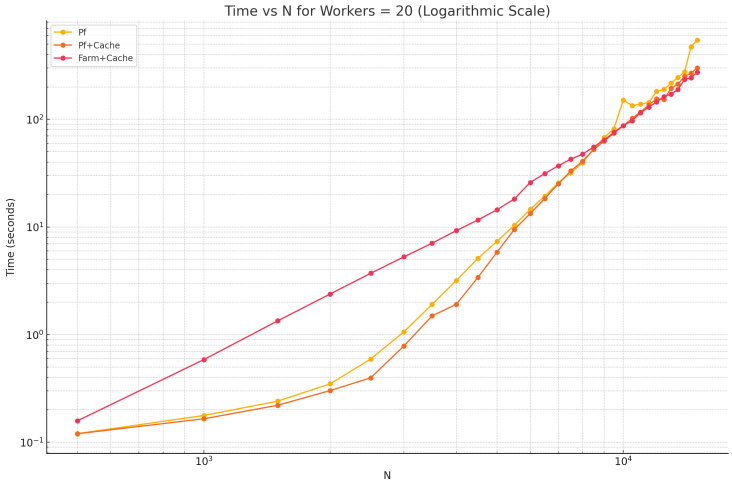

Figure 10: Performance comparison: Numa with 20 workers

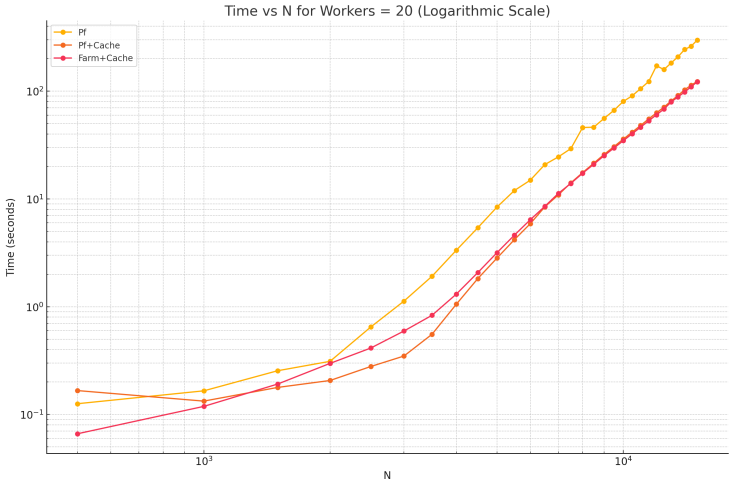

Figure 11: Performance comparison: Intel machine with 20 workers

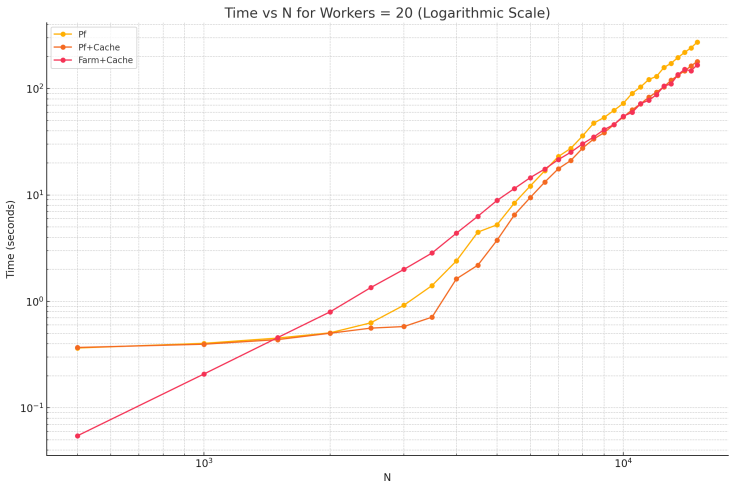

Figure 12: Performance comparison: Nvidia DGX with 20 workers

The graphs reveal some common patterns but also highlight significant differences. Notably, in the graph [11], the Farm method exhibits a logarithmic trend compared to the other two graphs, where its behavior is much more linear. Both the `parallel_for` with and without cache-oblivious algorithms demonstrate logarithmic trends. On NUMA systems, they tend to outperform the Farm method from the very beginning. As N increases, the Farm method shows a general improvement across all three machines, becoming the fastest solution for larger matrices (though by a small margin). A particularly notable behavior is observed in the graph [12], where the Farm method is significantly faster than the other methods for the initial values of N, before eventually aligning with them as N grows larger.

5 Distributed solution (MPI)

In the distributed version, the main process is separated from the worker processes, with the primary goal of managing and distributing tasks to various worker nodes. Each process maintains it's own local copy of the matrix to be computed. While this approach simplifies the process of updating worker nodes with the computed results via the main process, it comes at a significant cost in terms of memory consumption.

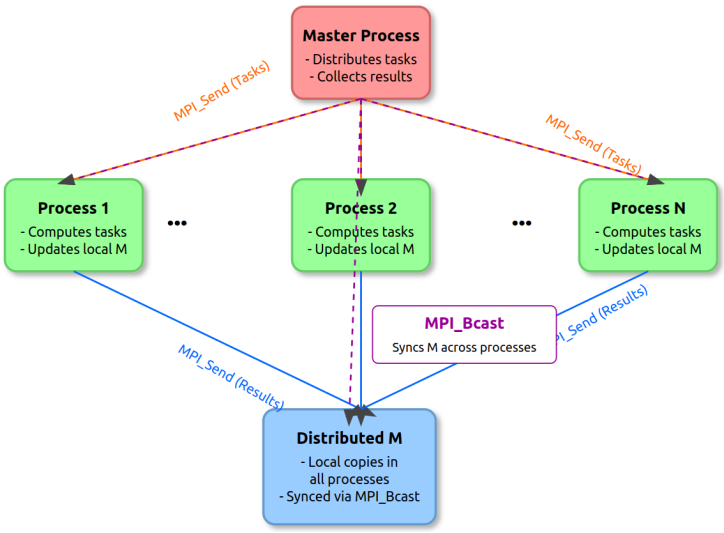

Figure 13: Distributed solution structure

One of the main challenges with this implementation is the memory overhead, as duplicating the matrix across multiple processes can be highly inefficient. Furthermore, the communication overhead between the main process and worker nodes, which is far from trivial, must be carefully managed to avoid bottlenecks. These factors contribute to the overall complexity and performance trade-offs of the distributed solution. Effective coordination and optimization of both memory usage and inter-process communication are critical to achieving scalability and efficiency in such systems.

5.1 Results

Using the same matrix size ($N = 4000$) across different configurations to evaluate how the distributed version of the program performs on varying numbers of nodes. The goal was to see the impact of spreading different tasks number over more or fewer nodes.

On below graph it's possiible observe how factors like inter-node communication and resource allocation affected execution time. This helps to assess the scalability of the distributed approach and identify potential bottlenecks as the system expands to more nodes.

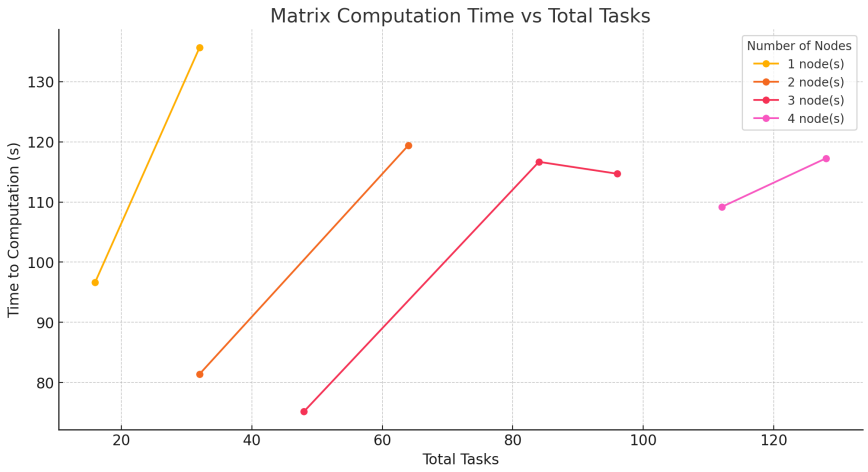

Figure 14: Different task and node distribution on the same problem

The graph [15] shows the impact on execution times for very large tasks. This test show us the performance advantage (and not) of distributing a task across multiple nodes and how this positively affects performance as the matrix size increases.

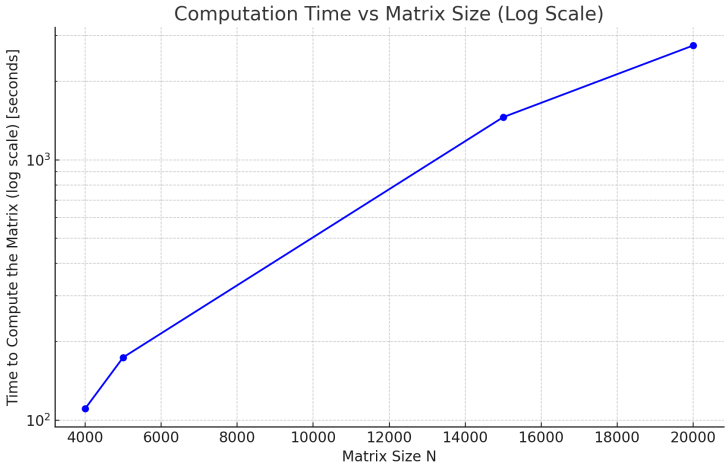

Figure 15: MPI version benchmark