SPM Project Report - Wavefront Computation (Project 1)

SPM 2023/24 - University of Pisa

Nicola Emmolo

Abstract

This report presents the implementation and analysis of various approaches for the parallelisation of an algorithm following a wavefront computation pattern on a square matrix. The main objective was to improve efficiency and performance through the use of different libraries and parallelisation techniques, including C++ with threads, FastFlow, OpenMP and MPI. The results obtained were evaluated in terms of execution time, speedup, efficiency and scalability. Detailed analyses were conducted on a single multi-core machine and on a cluster of multi-core machines. Finally, the strengths and weaknesses of each approach were identified, with a focus on scalability and efficiency based on the size of the array and the number of threads or nodes used.

1. Introduction

The issue addressed in this project concerns the parallelisation of an algorithm that follows a *wavefront* computation pattern on a square matrix of size $N \times N$. In this context, *wavefront* computation refers to a dependency model where the elements of a matrix are computed following a diagonal sequence, starting from the main diagonal (diagonal $k = 0$) and continuing up to the diagonal $k = N - 1$. Each element on a specific diagonal can be computed independently of the other elements of the same diagonal, but it cannot be computed until all the elements of the previous diagonal have been completed. This model reflects a computational flow that progresses through the matrix from the main diagonal to the upper right.

1.1. Algorithm Description

The algorithm employs the *dot product* operation to compute each element $e^k_{i,j}$ of the matrix M on a specific diagonal k. Specifically, for each element on the specified diagonal k, the value of $e^k_{i,j}$ is calculated as the result of a dot product between two vectors:

- The vector v^k_m, which contains the elements of row m of matrix M.
- The vector v^k_{m+k}, which contains the elements of column $m + k$ of matrix M.

The computation can be formalized as:

$$e^k_{i,j} = \text{dotproduct}(v^k_m, v^k_{m+k})$$

Where the dot product between vectors $v1$ and $v2$ is defined as:

$$\text{dotproduct}(v1, v2) = \sum_{i=1}^{k} v1[i] \times v2[i]$$

This operation is repeated for all elements of each diagonal of the matrix, with k varying from 1 to $N - 1$.

1.2. Objectives

The principal objective of the project is the implementation of parallel versions of the *wavefront* computation algorithm:

1. **Parallel Version on a Single Multicore Machine:** The algorithm has been parallelised in order to optimise the utilisation of the resources available on a single multicore machine. The utilisation of multiple *threads* enables the concurrent computation of matrix diagonal elements, thereby reducing execution times in comparison to the sequential version.
2. **Parallel Version on a Cluster of Multicore Machines:** The algorithm is extended to operate on a cluster of machines, with the objective of distributing the workload across multiple nodes and thereby improving performance.

2. Parallelizzation (C++ and Threads)

A principal approach to parallelising the code entails the utilisation of C++'s native thread model to perform simultaneous calculations. The use of threads allows for the division of the calculation of matrix elements along the upper diagonal, enabling the concurrent processing of multiple elements and resulting in enhanced performance compared to the sequential version.

In the code, `std::barrier` is employed for the purpose of synchronising the execution of threads during the processing phase. A barrier is a synchronization construct that blocks a certain number of threads until all participating threads reach a specific point in the program, known as a "synchronization point". Upon arrival of all threads, they are released and may then proceed with their execution. This is particularly useful in the context of *wavefront* computation, where threads must await the completion of the computation on a given diagonal before proceeding to the subsequent diagonal.

The `ThreadPool` is a mechanism that allows for the management of a set of reusable threads, avoiding the overhead of frequently creating and destroying threads. In the context of the code, the `ThreadPool` is used in the dynamic parallel version of the *wavefront* computation. Tasks are added to the thread pool queue, and threads in the pool execute the tasks in a dynamic manner. This means that tasks can be distributed among available threads based on their availability, improving the efficiency of resource utilization.

2.1. Implementation

The code implements two parallel versions of the *wavefront* computation:

1. **Static Parallel Version (`wavefront_parallel_static`)**:
   In this version, threads are assigned to compute specific elements of the matrix based on their ID. Each thread is responsible for processing a fixed portion of the matrix elements (static) and waits for all other threads to complete the computation of a given diagonal before proceeding to the next. At the conclusion of each diagonal, threads are synchronised using `std::barrier`.
2. **Dynamic Parallel Version (`wavefront_parallel_dynamic`)**:
   This version uses a `ThreadPool` to facilitate the dynamic assignment of tasks to threads . Tasks are added to the `ThreadPool` queue, and threads execute the tasks as they become available. Once more, `std::barrier` is used to synchronize threads, although the distribution of the workload is more flexible than in the static version.

3. Parallelizzation (FastFlow)

A second strategy for parallelizing the code uses the FastFlow library, which provides an efficient framework for parallel programming. FastFlow has been developed with the specific objective of fully exploiting multi-core architectures by offering high-performance primitives for parallelism. In the context of this project, FastFlow is used to simultaneously compute matrix elements along the diagonal.

In the code, FastFlow's `ParallelFor` class is used to implement parallelism. This class facilitates parallel execution of a `for` loop by automatically dividing tasks among available threads. This enables the efficient control of parallelism, obviating the necessity for manual thread management.

3.1. Implementation

As before, the code implements two parallel versions of the *wavefront* computation using FastFlow:

1. **Static Parallel Version with FastFlow (`wavefront_parallel_static_ff`)**:
   In this version, parallelism is managed through FastFlow's `parallel_for` method with static scheduling. The upper diagonal elements of the matrix are evenly divided among the threads, with each thread assuming responsibility for a fixed portion of the elements (static). This approach ensures that each thread has a similar workload, reducing wait times between threads and improving overall efficiency.
2. **Dynamic Parallel Version with FastFlow (`wavefront_parallel_dynamic_ff`)**:
   In this version, task distribution is managed dynamically using FastFlow's `parallel_for` method with a different scheduling mode. Tasks are assigned to threads dynamically based on their availability. This allows for greater flexibility in workload management, particularly useful when there are significant differences between the number of threads and the number of elements to be computed along the current diagonal. Again, FastFlow handles thread synchronization and management, thereby guaranteeing the optimal utilisation of available resources.

4. Parallelizzation (OpenMP)

Another parallelization strategy adopted for the computation of the *wavefront* pattern is the use of OpenMP, a parallel programming library for C++ that allows the creation of multi-threaded applications in a simpler and more manageable way. OpenMP provides compiler directives that enable the parallelization of code portions without the need for explicit thread management.

In this version as well, OpenMP is used to parallelize the computation of the elements along the diagonal. The OpenMP runtime automatically creates and manages threads, distributing the workload among the available threads.

4.1. Implementation

As before, the code includes two parallel versions of the *wavefront* computation using OpenMP:

1. **Static Parallel Version with OpenMP (`wavefront_parallel_static_omp`)**:
   In this version, the `#pragma omp parallel for` directive with `schedule(static)` is used to distribute the upper diagonal elements evenly among the threads, ensuring that the workload is uniformly distributed and predictable.
2. **Dynamic Parallel Version with OpenMP (`wavefront_parallel_dynamic_omp`)**:
   In this version, the `#pragma omp parallel for` directive with `schedule(dynamic)` is used. The dynamic task distribution permits the assignment of tasks as they become available, dynamically balancing the workload.

5. Results: Single Multi-Core Machine

In order to assess the efficacy of the various parallelisation strategies presented, several tests were conducted using the implemented code. The experiments aimed to measure the execution times, the speedup, the efficiency and scalability (strong and weak) of each version of the wavefront computation, comparing static and dynamic task scheduling on various parallel frameworks. Additionally, a sequential version of the algorithm was tested as a baseline for comparison.

The tests were conducted on matrix sizes ranging from 128×128 to 4096×4096 elements and using a varying number of threads (from 1 to 40, increasing by steps of 2). Each configuration was repeated 10 times to ensure accuracy and to account for any potential fluctuations in the execution times.

5.1. Sequential

In the graph [1], the sequential execution time of the algorithm as a function of the matrix size $N \times N$ can be seen. Clearly, as the matrix size increases, the execution time grows significantly, and for $N = 4096$, it reaches up to 90 seconds, demonstrating that the computational complexity of the algorithm increases rapidly with the matrix size.

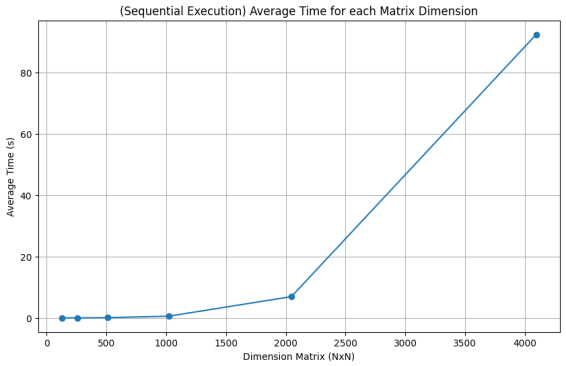

Figure 1: Sequential execution time as the matrix size increases

5.2. C++ and Threads

In these graphs [2], the **execution time** for static and dynamic scheduling can be observed as the matrix size changes and as a function of the number of threads used (from 1 to 40). The improvement compared to sequential execution is evident. For a large matrix, the execution time drops significantly as the number of threads increases. However, beyond a certain number of threads (around 10-20), the improvement becomes less significant, showing a tendency towards stabilization. For smaller matrices, the gain in terms of time is minimal, indicating that the overhead associated with parallelism management outweighs the benefits of parallelization on small sizes. It can also be noted that, for $N = 4096$ with dynamic scheduling, the execution time initially decreases and then slightly rises, suggesting that there may be greater overhead compared to static scheduling in certain scenarios. In fact, in this case, static scheduling seems to offer slightly better performance than dynamic scheduling, especially with a larger number of threads.

Regarding the **speedup** [3], for large matrix sizes and static scheduling, the speedup initially increases with the number of threads, which is expected behavior in effective parallelization. However, for smaller matrix sizes, the speedup seems to stabilize or even slightly decrease. This may be due to the fact that the overhead of thread management becomes significant compared to the benefits of parallelization for smaller problems. On the other hand, considering dynamic scheduling, the speedup obtained is lower compared to the static version, particularly for large matrices. Moreover, for large matrices, the initial trend shows a peak followed by a decline or stabilization, while for smaller matrices, the speedup generally remains low.

Moving on to **efficiency** [4], it is relatively high for both versions when using a few threads, especially for larger matrices. This indicates that, with light parallelization, there is good resource

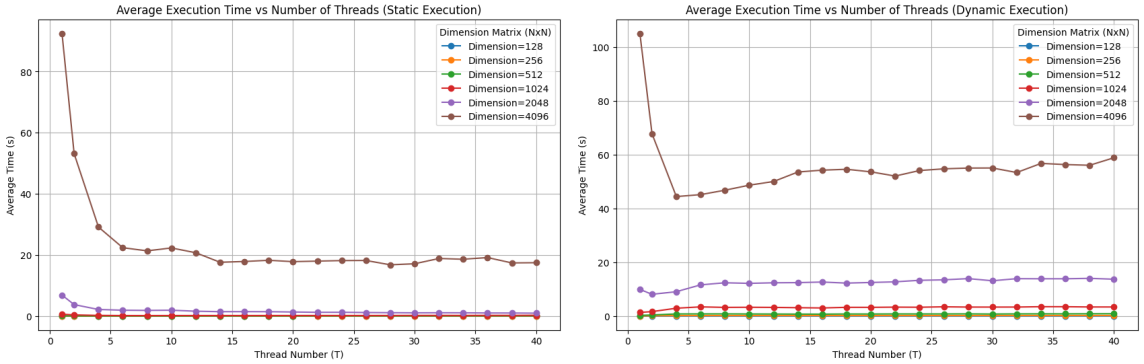

Figure 2: Execution times for static and dynamic scheduling, as the number of threads increases

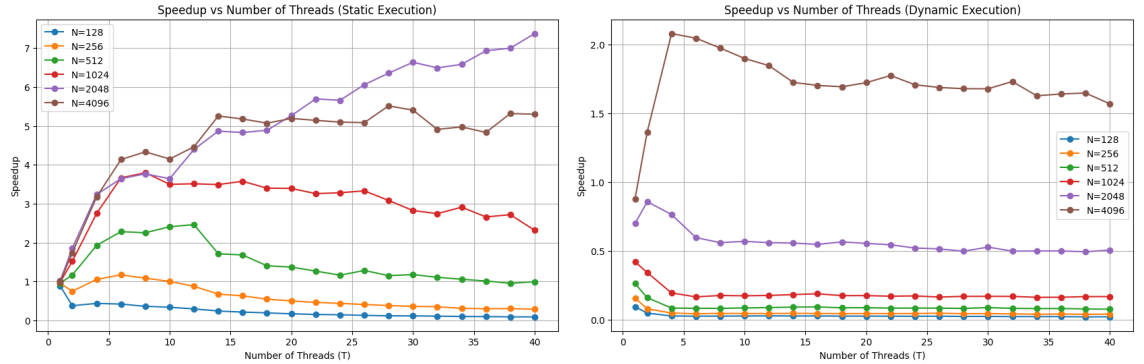

Figure 3: Speedup for static and dynamic scheduling, as the number of threads increases

utilization. Both methods show a rapid drop in efficiency as the number of threads increases, with dynamic scheduling showing an even more pronounced decline. The additional overhead for dynamically managing threads seems costly in terms of resources. With a high number of threads, the efficiency tends to stabilize at low values for all matrix sizes, reflecting the increase in overhead compared to the benefits of parallelization.

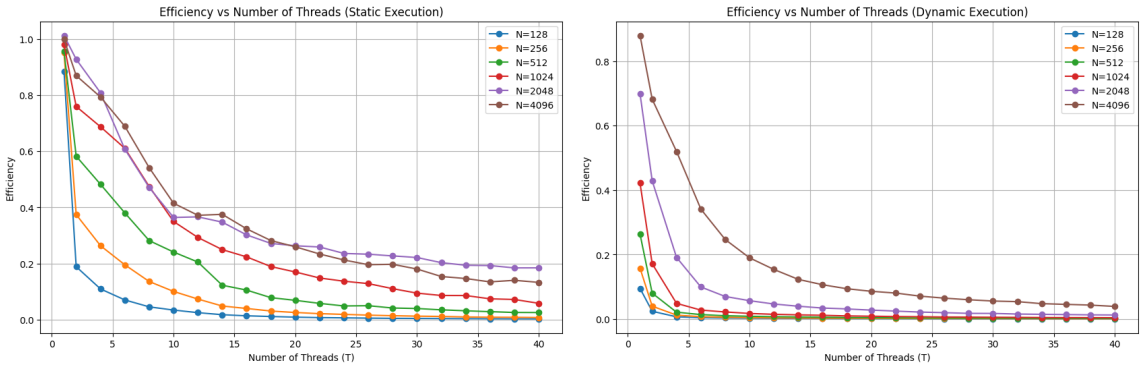

Figure 4: Efficiency for static and dynamic scheduling, as the number of threads increases

Considering **scalability** [5], regarding strong scalability, a similar trend to speedup can be observed, where for larger matrices, scalability progressively increases with the number of threads, while for smaller problems, scalability remains very low. In this case as well, dynamic execution has worse scalability compared to static execution. Indeed, it can be seen that scalability is limited even for large sizes.

Regarding weak scalability, in both cases (static and dynamic), it can be observed that scalability drops drastically. There are no significant differences between the two execution modes, and neither shows good results for larger matrices.

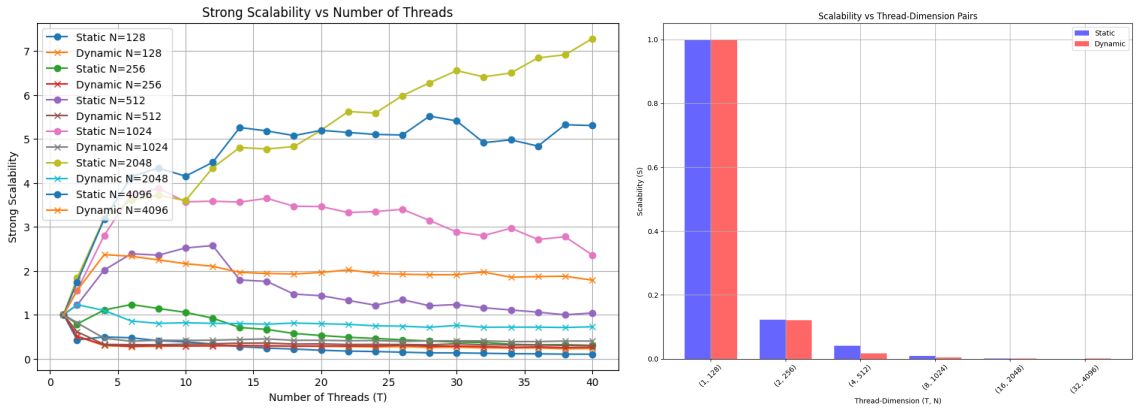

Figure 5: Strong and weak scalability for static and dynamic scheduling

5.3. FastFlow

In these graphs [6], the **execution time** for static and dynamic scheduling can be observed, but this time utilizing FastFlow. Analyzing static scheduling, for a limited number of threads, the execution time decreases rapidly. However, beyond 10 threads, there is an increase in execution time, especially for larger matrix sizes. Looking at the results with dynamic scheduling, the observations are similar to static scheduling, but dynamic scheduling seems to be more robust with a higher number of threads. However, while the execution time tends to increase beyond 30-40 threads, the increase is not as pronounced as in the static version. Overall, FastFlow still provides better execution times compared to the previous version.

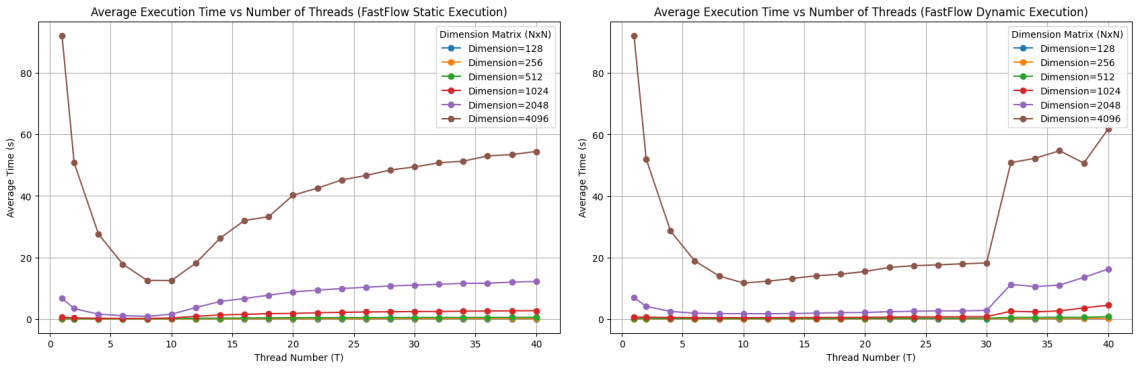

Figure 6: FastFlow execution times for static and dynamic scheduling, as the number of threads increases

Regarding **speedup** [7], the graphs show that for small or medium matrix sizes, the speedup trend is rather flat and remains around very low values, indicating that parallelization does not bring significant benefits. On the other hand, for larger sizes, an initial increase in speedup can be observed, followed by dips. Unlike static scheduling, dynamic scheduling shows a more gradual decline, and the speedup remains higher, indicating better load distribution management.

Regarding **efficiency** [8], static scheduling shows less efficient behavior as the number of threads

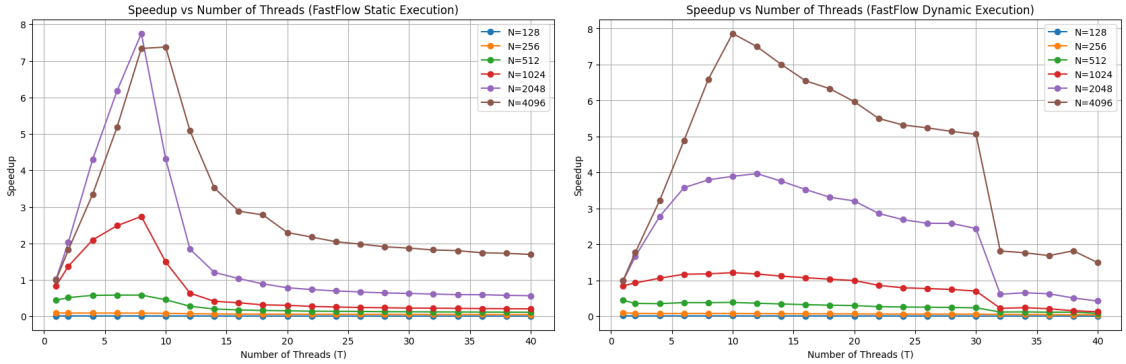

Figure 7: FastFlow speedup for static and dynamic scheduling, as the number of threads increases

increases, especially for large matrix sizes. Dynamic scheduling offers better load balancing and maintains efficiency with a higher number of threads, particularly for large matrix sizes. However, in both cases, once the optimal number of threads is exceeded, efficiency decreases, with a more pronounced decline in static scheduling.

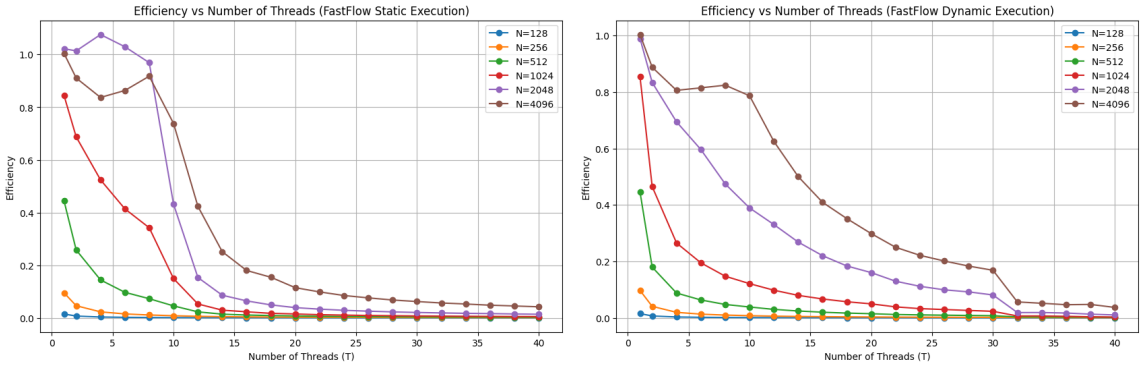

Figure 8: FastFlow efficiency for static and dynamic scheduling, as the number of threads increases

Moving on to **scalability** [9], regarding strong scaling, we observe that static scheduling shows a steeper initial increase in scaling compared to dynamic scheduling, but also a faster drop after a certain number of threads. Dynamic execution, on the other hand, maintains more stable performance with a larger number of threads.

In the weak scaling graph, it can be seen that for small sizes, both the static and dynamic models show ideal behavior. However, efficiency starts to decline afterward. The static model offers slightly better efficiency compared to the dynamic one in weak scaling scenarios.

5.4. OpenMP

In these graphs [10], the **execution time** for static and dynamic scheduling can be observed, utilizing OpenMP. Both parallel versions, static and dynamic, drastically reduce execution times compared to the sequential version and previous parallel versions. In this specific context, there does not seem to be a substantial difference, but dynamic scheduling might offer slight flexibility in scenarios with more variable workloads.

Moving on to **speedup** [11], in both scheduling cases, with medium or large matrices, the speedup increases rapidly as the number of threads grows. In situations where the workload is evenly distributed,

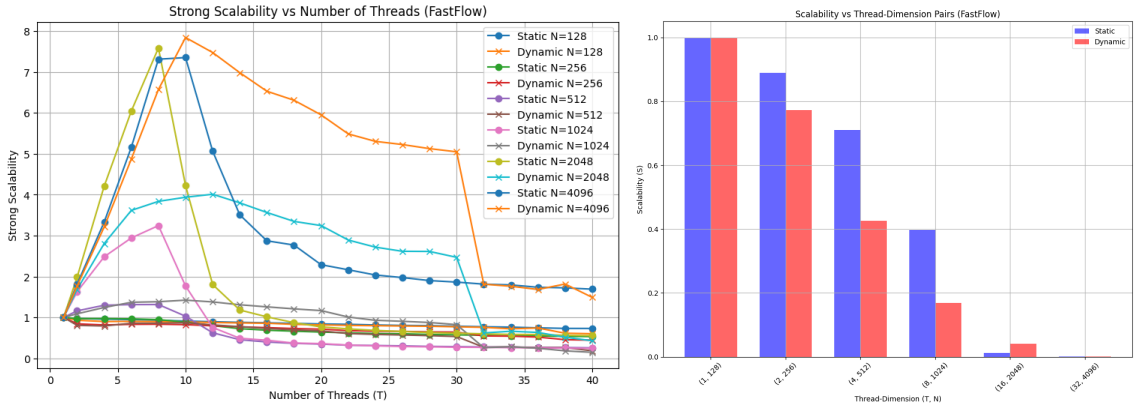

Figure 9: FastFlow strong and weak scalability for static and dynamic scheduling

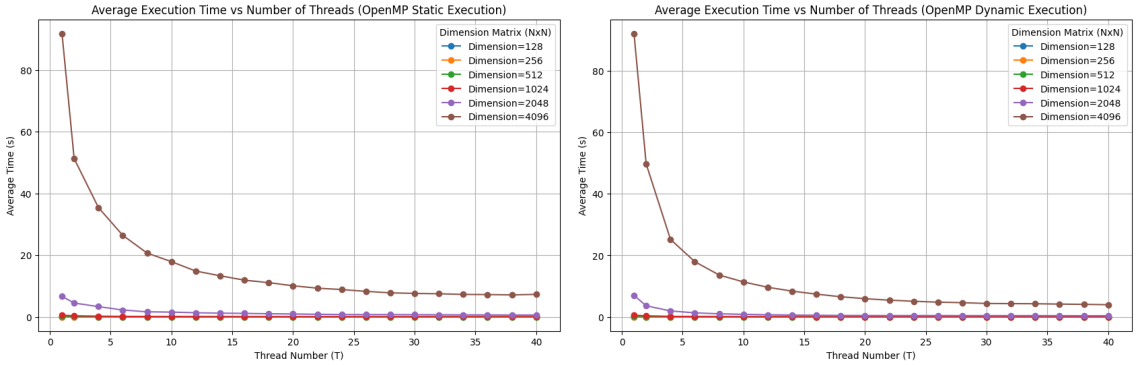

Figure 10: OpenMP execution times for static and dynamic scheduling, as the number of threads increases

static scheduling tends to be more efficient. In fact, the overhead introduced by dynamic management can result in slightly lower speedup compared to static scheduling with a smaller number of threads. However, in cases of unbalanced workloads, dynamic scheduling can maintain more consistent speedup as the number of threads increases.

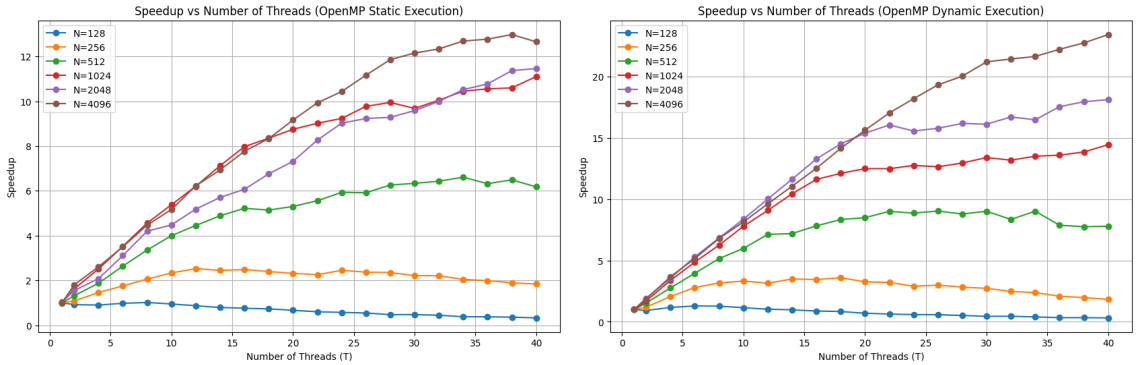

Figure 11: OpenMP speedup for static and dynamic scheduling, as the number of threads increases

Considering **efficiency** [8], there is generally better efficiency compared to the previous cases. However, static scheduling shows a rapid initial drop in efficiency, which stabilizes at a very low value, especially for small matrices. On the other hand, dynamic scheduling decreases more gradually, showing better workload management, particularly with larger matrices.

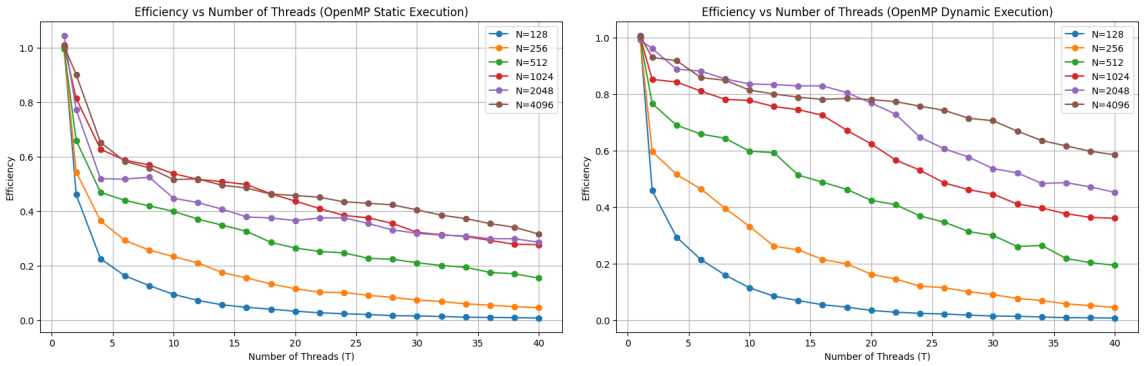

Figure 12: OpenMP efficiency for static and dynamic scheduling, as the number of threads increases

Regarding **scalability** [13], what can be said about strong scalability is that static scheduling increases with a growing number of threads but tends to stabilize earlier than in the dynamic case, particularly for smaller matrices. On the other hand, strong scalability in dynamic scheduling is more pronounced, with a steady increase even with many threads.

As for weak scalability, a rapid decrease in scalability can be observed as the number of threads and the matrix size increase, for both static and dynamic methods.

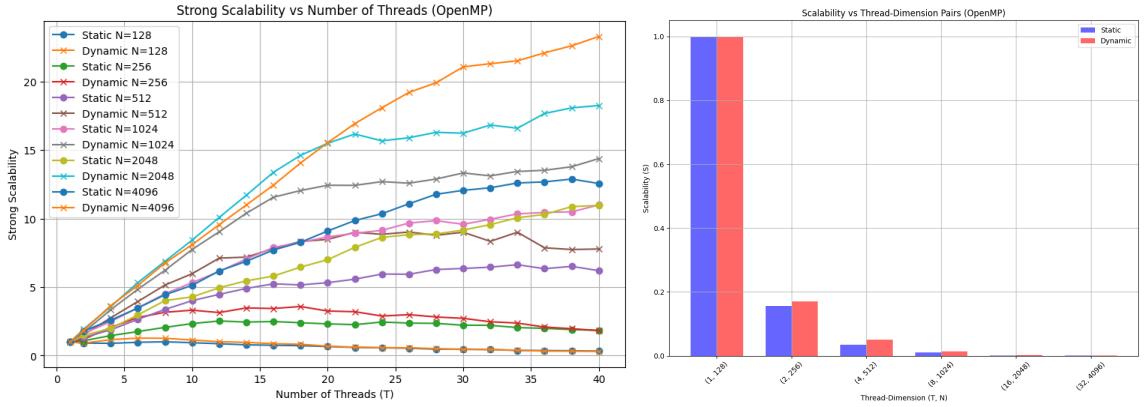

Figure 13: OpenMP strong and weak scalability for static and dynamic scheduling

6. Parallelization (Message Passing Interface)

An alternative approach to parallelizing the *wavefront* pattern computation is the use of MPI (Message Passing Interface), a library for distributed parallel programming, commonly used in distributed memory systems such as computing clusters. MPI allows for the division of work among multiple nodes, each of which performs a portion of the computation, communicating with each other through message passing.

In this implementation, MPI is used to divide the computation among multiple processes, each responsible for a part of the work, with a synchronization mechanism to ensure that the data remains correct during parallel processing.

6.1. Implementation and Execution

The code implements a parallel version of the *wavefront* computation using MPI:

1. **Parallel Version with MPI (`wavefront_parallel_mpi`)**:
   In this version, the computation of elements along the upper diagonal of the matrix is divided among MPI processes. Each process performs a part of the work, and the processes synchronize with each other using an MPI barrier (`MPI_Barrier`) after each step. The assignment of work to each process is based on the process rank, with the row index modulo operation. This technique ensures that workloads are evenly distributed among the processes, making the best use of available resources.

7. Results: Cluster of Multi-Core Machines

Also in this case, to evaluate the effectiveness of the parallel implementation of the *wavefront* algorithm with MPI, a series of tests were conducted using different parallelization scenarios. The tests were performed on a cluster of nodes with varying configurations, utilizing different numbers of nodes to analyze the impact on performance and execution time.

The tests were conducted on matrix sizes ranging from 128×128 to 4096×4096 elements and using a varying number of nodes (1, 2, 4, 6 and 8). Each configuration was repeated 10 times to ensure accuracy and to account for any potential fluctuations in the execution times.

In these graphs [14], the **execution time** for static and dynamic scheduling can be observed, utilizing MPI. A significant improvement in execution time is noted when moving from 1 node (about 19 s) to 8 nodes (about 3.7 s), with a reduction in execution time of over 80%. The reduction is very noticeable up to 4 nodes, beyond which the improvement becomes less pronounced. In this case, it appears that the algorithm is extremely efficient for large matrices. However, for smaller matrices, the overhead of MPI communication limits the benefits of parallelization, with marginal or no improvements compared to sequential execution.

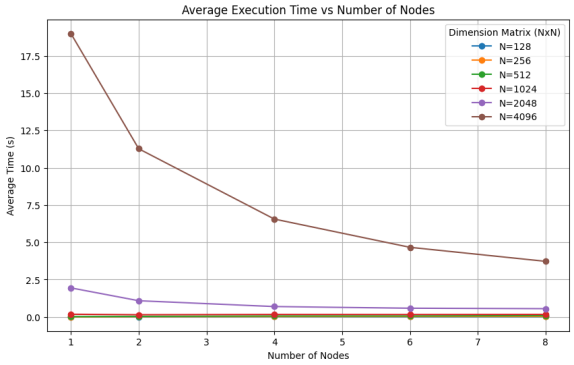

Figure 14: Message Passing Interface execution times, as the number of nodes increases

Moving on to **speedup** [15], it can be observed that there is a positive and nearly linear trend for large matrices, where a clear benefit of parallelization is evident. However, this trend shifts to negative or stagnating for smaller matrices, where increasing the number of nodes does not lead to a significant improvement.

Regarding **efficiency** [16], a positive trend can be observed for large matrices, where efficiency remains relatively high, although it decreases with the increase in nodes. On the other hand, a negative trend is seen for smaller matrices, where efficiency drops dramatically as the number of nodes increases.

Regarding **scalability** [17], in the case of strong scaling, a trend similar to speedup is observed, which is positive for large matrices, where the addition of more nodes improves performance almost linearly, and negative for small matrices, where the algorithm does not scale effectively.

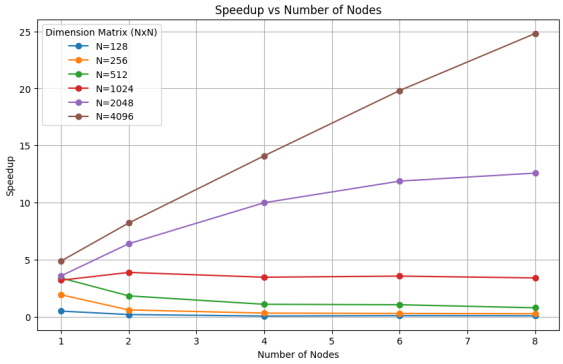

Figure 15: Message Passing Interface speedup, as the number of nodes increases

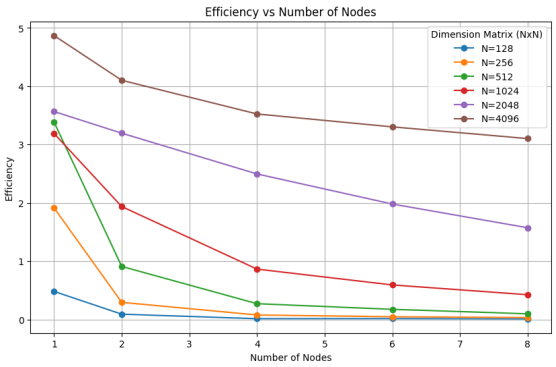

Figure 16: Message Passing Interface efficiency, as the number of nodes increases

In the case of weak scaling, the graph shows very low weak scalability. This demonstrates that the algorithm fails to maintain constant execution time as the problem size grows proportionally to the number of nodes, except for the smallest matrix.

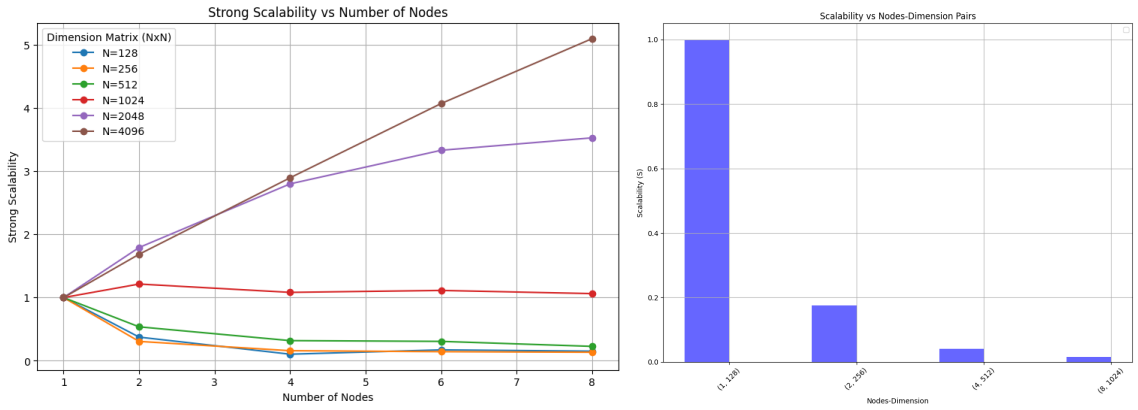

Figure 17: Message Passing Interface strong and weak scalability

8. Results: Comparison

The previous sections of the results examined the performance of the different parallelisation techniques on a single multi-core machine and on a cluster of multi-core machines. In the context of the single machine, parallel execution showed significant speed improvements over sequential implementation, especially for larger matrix sizes. However, the number of threads beyond a certain limit did not lead to

further improvements, suggesting that the overhead of thread management was beginning to outweigh the benefits of parallelisation. With regard to the cluster, MPI demonstrated excellent scalability for large arrays, but its communication overhead reduced efficiency for smaller arrays.

In terms of speedup, the static version tended to perform better than the dynamic version for both FastFlow and OpenMP, especially for balanced loads. Efficiency was higher in configurations with fewer threads, but decreased rapidly as the number of threads increased. MPI, while offering good scalability on a cluster, showed efficiency limitations when applied to small arrays. Finally, strong scalability was good for larger arrays, but weak scalability suffered in almost all cases.

9. Compilation and Execution

In regard to the compilation and execution of programs for the various versions, the code may be compiled using the Makefile located within the project folder. Once the compilation process is complete, the programs can be executed by entering the optional parameters directly on the command line:

- `./UTW [N] [T] [mode] [log_file_name]`
- `./UTWFF [N] [T] [mode] [log_file_name]`
- `./UTWOMP [N] [T] [mode] [log_file_name]`
- `mpirun -np [nodes] ./UTWMPI [N] [log_file_name] [nodes]`

Where `N` is the size of the square matrix, `T` the number of threads to use, `mode`: the execution mode, `log_file_name` the name of the file where execution results are saved, and `nodes` for the number of nodes to use. Note that the `mode` can be `'s'` (only for UTW) for sequential execution, `'ps'` (not for UTWMPI) for a parallel execution with static scheduling, and `'pd'` (not for UTWMPI) for a parallel execution with dynamic scheduling.

10. Conclusions

During the development of the project, several technical challenges emerged. The efficient management of threads and computing resources was complex, especially with approaches such as dynamic parallelisation, which introduced significant overhead. Solving these problems involved optimising thread and resource management techniques, with a focus on reducing synchronisation overhead. It was also noted that although parallelisation can lead to significant performance improvements, the choice of the right technique and configuration is crucial to achieve the best results.

It should also be added that the timing and limited access to the remote cluster was a further obstacle, preventing me from easily performing further tests with MPI. In general, the solutions adopted proved to be satisfactorily effective, but further optimisations could improve the results.