

Distributed Wavefront Computation

Giulia Ghisolfi, g.ghisolfi@studenti.unipi.it, 664222

Master Degree in Computer in Science (Artificial Intelligence Curriculum), University of Pisa

Parallel and Distributed Systems: Paradigms and Models course (305AA), Academic Year: 2023-2024

1 Problem Introduction

This report presents several parallelization strategies to optimize wavefront computation. The wavefront problem, characterized by its upper-triangular computation pattern, requires diagonal elements to be computed sequentially, while elements within the same diagonal can be computed independently.

This project implements and analyzes both multi-threaded and multi-process parallelization. The single-process implementations were developed using the **FastFlow** library, following two main approaches: data parallelism, utilizing a parallel-for implementation with both static and dynamic task scheduling, and stream parallelism, employing the farm model for task distribution. The multi-process implementation was designed for a cluster of multi-core machines, leveraging the MPI framework to enable distributed computation. Additionally, a sequential implementation was developed and used as a baseline for performance comparison.

2 Parallelization Strategy

2.1 MultiThread Implementation

For the multi-threaded version, we implemented the solution using the **FastFlow** library. The wavefront computation follows a sequential approach for diagonals, where each diagonal is processed one after the other. However, within each diagonal, the computation of elements is fully parallelized, distributing the workload evenly among threads.

The parallelization strategy is applied to both the main diagonal, where all elements are initialized in parallel, and the upper diagonals, where elements are computed concurrently.

Data parallelism

The first parallelization approach explored data parallelism. This technique used a parallel-for implementation to distribute the diagonal elements among different workers.

Tasks are created by grouping elements of the diagonal being computed into chunks of size G . The chunk size parameter allows fine-tuning of the workload assigned to each thread, effectively controlling the granularity of task assignment. Task distribution follows two distinct scheduling strategies: static and dynamic.

For the static scheduling approach, we used the `parallel_for_static` class. This method ensures that the workload is assigned to workers before computation begins. Each thread processes a fixed number of elements, leading to a predictable and evenly balanced distribution. This implementation is simple and efficient when the workload is uniform and predictable. However, a potential limitation arises from load imbalance if some workers complete their tasks earlier than others, leading to idle time and reduced overall efficiency.

To overcome this limitation, we used the `ParallelFor` class. In this approach, each worker is dynamically assigned up to chunk iterations at a time. Tasks are distributed dynamically among workers during execution, allowing for better workload balancing.

Stream parallelism

We also implemented a stream parallelism approach using the `ff_farm` class from the `FastFlow` library. The farm model partitions the problem into subtasks and assigns them to worker threads until the entire wavefront computation is completed. The `ff_farm` model follows a master-worker architecture, where an `emitter` assigns tasks to a pool of `workers` that execute them in parallel.

The `emitter` partitions each diagonal into subtasks of size G , dispatches them to the workers via the `ff_send_out` method. The chunk size parameter G determines the granularity of task distribution. Additionally, it gathers feedback signals from the workers through a feedback channel created using the `wrap_around` function, allowing it to track the progress of the current diagonal's computation. Once all elements of a diagonal have been processed, the emitter assigns the tasks for the next diagonal to the workers, repeating this process until the entire matrix has been computed.

Each `worker` processes its assigned chunk of elements by computing the dot product and storing the result in the matrix. After completing its assigned computation, sends a completion signal back to the emitter using `ff_send_out`. This signal allows the Emitter to track how many tasks have been completed for the current diagonal.

2.2 MultiProcess Implementation

For the multi-process version of the wavefront computation, MPI is used to distribute the workload efficiently.

The task distribution follows a block partitioning strategy, where each process is assigned a portion of the diagonal to compute. This approach ensures balanced workload allocation, with the last process handling extra elements when the total number of diagonal elements is not evenly divisible among processes.

Each MPI process computes its assigned diagonal elements locally, using `OpenMP` to further parallelize the computation of individual elements. Results are stored locally, ensuring efficient hybrid parallelism.

Once computation is complete, each MPI process shares its results using `MPI_Allgatherv`, which aggregates all partial results into a global result. Before moving to the next diagonal, each process updates its local matrix using the global result. This update is fully parallelized across all elements of the diagonal using `OpenMP`. This sequence of operations ensures the efficient synchronisation of data and its storage.

3 Experimental Setup

To evaluate the performance of each strategy, tests were run under different configurations. These tests measured execution times, providing insight into scalability, speed-up and efficiency.

Execution times were only measured after the matrix initialisation phase. Measurements covered wavefront computation from start to finish. For each test, multiple executions were con-

ducted to mitigate inaccuracies caused by temporary performance fluctuations. The reported values are the arithmetic mean of these. All reported times are expressed in milliseconds.

The tests for the different parallelization strategies, as well as the results for the sequential baseline, used as a reference for comparison, were conducted on *spmcluster.unipi.it*, referred to as *spmcluster*, from this point.

4 Performance Analysis: MultiThread Implementation

In the following section, the performance analysis of the multi-thread implementation will be examined. In all three proposed versions, the key parameters were the dimension of the matrix N , the granularity G (the chunk size), and the number of threads T used for parallelization. The experiments were performed on a single node of the *spmcluster*.

4.1 Strong Scalability

In order to perform a robust scalability analysis, a series of tests were conducted on a 2048×2048 matrix, evaluating each implementation with different granularity values ($G = 1, 16, 32, 128$).

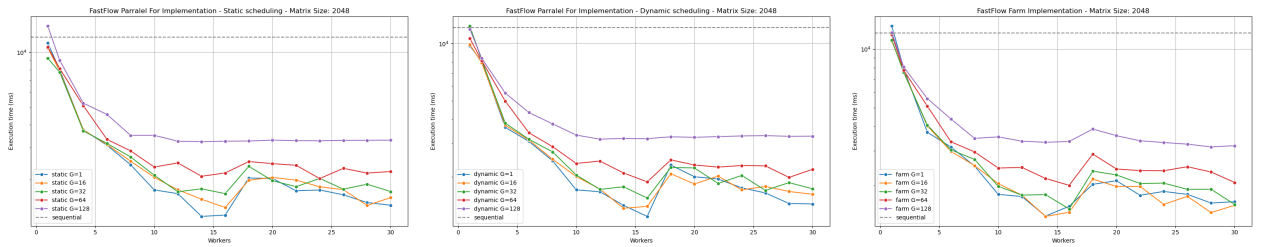


Figure 1: Execution times for the strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 and different granularity values ($G = 1, 16, 32, 128$). The number of workers ranges from 1 to 30. From left to right: parallel-for implementation with static scheduling, dynamic scheduling, and the farm model. The results from the sequential baseline for the corresponding matrix sizes are shown in the plots as dashed lines.

Figure 1 presents the execution times for the various parallelization strategies. Consistent performance trends are observed across all configurations as the number of workers and granularity settings vary. Performance improves with decreasing chunk size, suggesting that smaller tasks lead to better workload distribution and parallel efficiency.

For the parallel-for implementation, the best performance is achieved with $G = 1$, regardless of the chosen scheduling strategy, as it ensures optimal load balancing and minimizes task management overhead. With static scheduling and $G = 1$, each thread receives an almost identical workload, leading to a well-balanced execution without requiring further adjustments. Dynamic scheduling allows workers to request new tasks as they finish old ones, reducing workload imbalances. However, since $G = 1$ already provides an evenly distributed workload, the advantage of dynamic scheduling diminishes, making its behavior similar to static scheduling.

For the farm implementation, the best results are obtained with $G = 16$, though the difference compared to stochastic task scheduling is minimal. The slight advantage of $G = 16$ over stochastic scheduling in the farm implementation is likely due to a better balance between parallelism and communication overhead. At very fine granularity, the sender must frequently

send small tasks to the workers, increasing communication costs and synchronisation overhead. Frequent task distribution leads to higher contention and delays, reducing efficiency. In contrast, with $G = 16$, workers receive larger task batches, reducing the number of messages exchanged. This reduces context switching overhead and ensures more efficient use of computational resources for better performance.

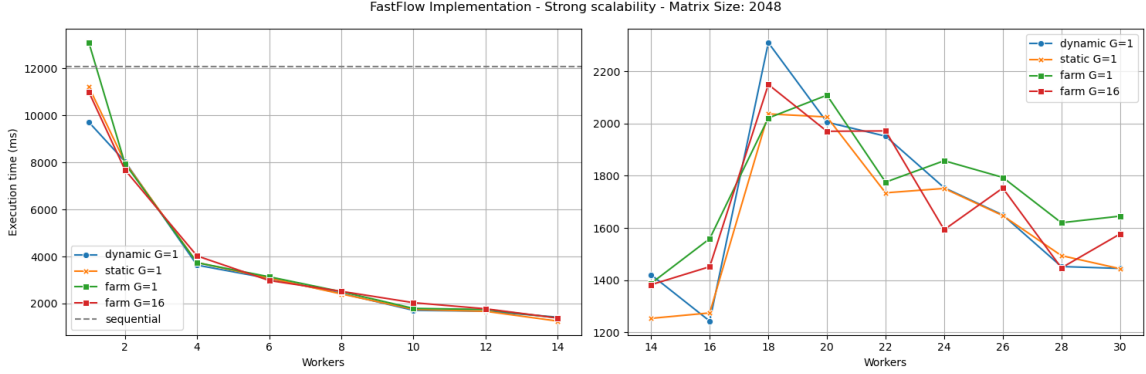


Figure 2: Strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 . The figure reports the execution times for the best-performing configuration of each implementation: parallel-for with static and dynamic scheduling (granularity $G=1$) and farm implementation (granularity $G=1$ and $G=16$). The left plots show execution times with a number of workers ranging from 1 to 14, while the right plots cover the range from 14 to 30, using different y-axis scales.

In Figure 2, only the best configurations for each implemented strategy are reported to facilitate a clearer comparison. In the parallel-for model, we observe that static scheduling performs slightly better than dynamic scheduling, as expected given that $G = 1$, where the workload is already well-balanced across workers.

In all strategies, the computation time decreases as the number of workers increases up to $W = 14$ (which corresponds to 15 active workers, including the master thread). However, beyond $W = 16$ (17 active threads in total), execution time increase. This behavior is likely due to the hardware configuration of the cluster node used for testing, which has 16 physical cores and 32 logical threads, with hyperthreading enabled.

From $W = 18$ onward, execution time starts decreasing again, suggesting the system partially compensates induced inefficiencies, through better utilization of logical cores and reduced contention effects, depending on configuration.

Only in the parallel-for strategy with dynamic scheduling sees execution time decrease up to $W = 16$, regardless of the chunk size G , before increasing in a manner similar to the other strategies. Dynamic scheduling better handles work distribution between physical and logical cores, allowing more efficient use of resources up to 16 workers. Unlike static scheduling, it redistributes tasks as workers complete them, which is likely to mitigate the early effects of resource contention.

For all strategies, when using the $G = 128$ configuration, execution time decreases up to $W = 8$ but then remains nearly constant due to load imbalance. Additionally, this configuration consistently delivers the worst performance across all tested worker counts. With $G = 128$, tasks are too large, limiting the number of parallel tasks and leading to poor resource utilization and increased idle time.

Speedup

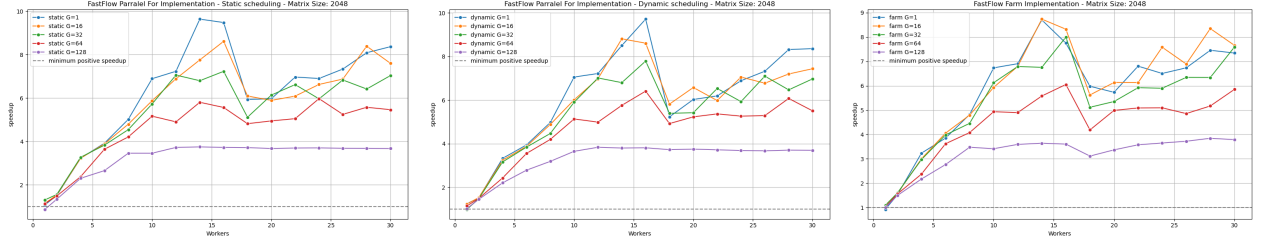


Figure 3: Speedup metrics for the strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 and different granularity values ($G = 1, 16, 32, 128$). The number of workers ranges from 1 to 30. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

From the speedup analysis is reported in Figure 3, we can deduce the following insights. In all three implementations, speedup increases linearly as the number of workers grows, up to $W = 16$. Beyond this point, there is a noticeable drop, consistent with previous findings that hyperthreading introduces contention rather than improving performance.

Consistent with previous observations, $G = 1$ achieves the highest speedup, confirming that smaller chunk sizes allow for better load balancing and parallel efficiency. Conversely, $G = 128$ performs the worst, as larger chunk sizes lead to reduced parallel efficiency and increased load imbalance.

Comparing the different strategies, static scheduling achieves the highest peak speedup, slightly outperforming dynamic scheduling.

Scalability

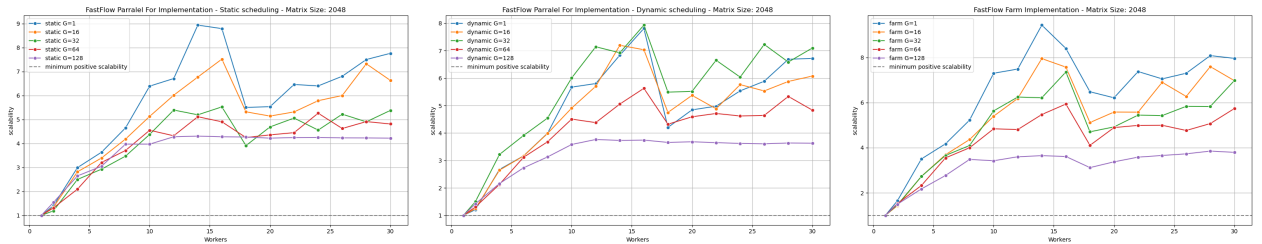


Figure 4: Scalability metrics for the strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 and different granularity values ($G = 1, 16, 32, 128$). The number of workers ranges from 1 to 30. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

Scalability, reported in Figure 4, follows a similar trend, reaching its peak at $W = 16$ before fluctuating due to hyperthreading effects. Parallel-for with static scheduling exhibits the best scalability, while the farm implementation shows higher variability, likely due to communication overhead in the emitter-worker model. After $W = 18$, performance stabilizes as logical threads are utilized more effectively.

For parallel-for implementation with dynamic scheduling, the configuration with $G = 32$ shows the highest scalability, even though its execution times are higher than those obtained

with lower G values. This is because the single-process version with $G = 32$ performs worse than the others, and even worse than the sequential implementation.

Efficiency

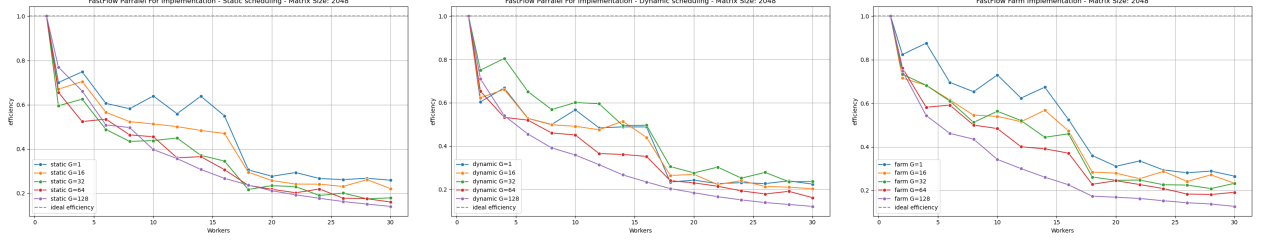


Figure 5: Efficiency metrics for the strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 and different granularity values ($G = 1, 16, 32, 128$). The number of workers ranges from 1 to 30. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

The efficiency analysis reveals a progressive decline as the number of workers increases, which is expected in strong scalability tests due to increasing overheads and resource contention. For all implementations, efficiency drops significantly beyond $W = 16$, aligning with the previous findings that hyperthreading introduces contention rather than improving performance.

Cost

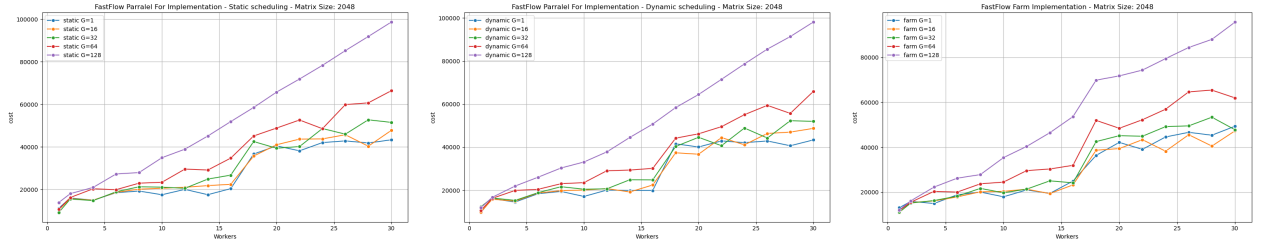


Figure 6: Cost metrics for the strong scalability analysis of the **FastFlow** implementations, tested with a matrix size of 2048×2048 and different granularity values ($G = 1, 16, 32, 128$). The number of workers ranges from 1 to 30. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

The cost metric, shown in Figure 6, provides insights into parallel efficiency and resource utilization. For all implementations, cost increases with the number of workers, indicating diminishing returns as more resources are allocated.

Tests performed with granularity $G = 128$ consistently exhibits the highest cost, confirming that large chunk sizes lead to inefficient workload distribution and poor scalability. Conversely, for lower G values, cost remains lower up to $W = 16$, reinforcing that smaller granularity improves parallel efficiency and optimizes resource consumption.

The farm implementation generally shows a slightly higher cost, likely due to communication overhead between the emitter and workers, which becomes more pronounced as the number of workers increases.

4.2 Weak Scalability

To evaluate weak scalability, the matrix size was increased proportionally to the number of workers used for parallelization. Since, as observed in the previous analyses, performance scales efficiently up to 14 workers (i.e., 15 threads in total, including the master thread), the weak scalability tests were conducted within this range. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840.

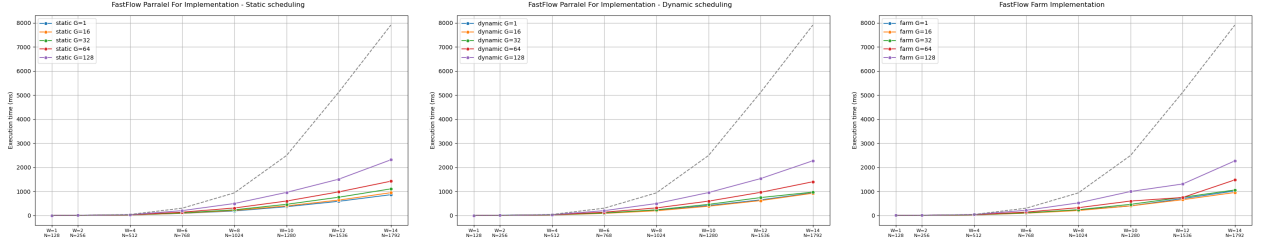


Figure 7: Execution times for the weak scalability analysis of the **FastFlow** implementations. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840. Tests are performed using different granularity values ($G = 1, 16, 32, 128$). From left to right: parallel-for implementation with static scheduling, dynamic scheduling, and the farm model. The results from the sequential baseline for the corresponding matrix sizes are shown in the plots as dashed lines.

The weak scalability results, reported in Figure 7, indicate that execution time increases in proportion to problem size, with the number of workers also growing.

Execution time scales exhibit a marginal increase for lower granularity values ($G = 1, G = 16$), indicating good weak scalability. Test performed with larger granularity values ($G = 128$) show significantly worse performance, confirming that larger task sizes lead to load imbalance. The results confirm that fine-grained task distribution improves weak scalability, while larger chunks and scheduling overheads negatively impact performance as the workload grows.

Speedup

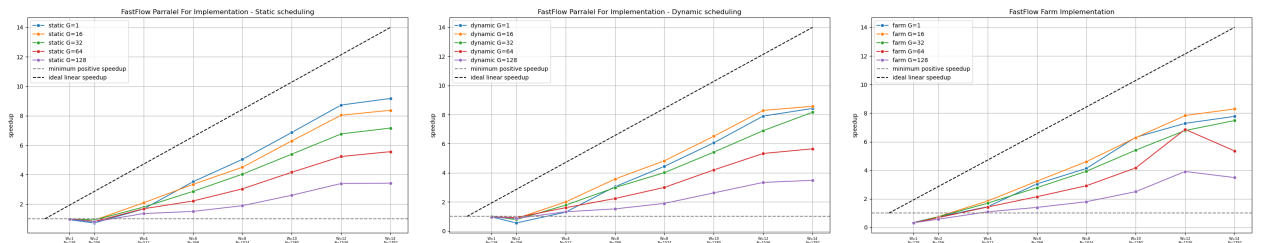


Figure 8: Speedup metrics for the weak scalability analysis of the **FastFlow** implementations. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

From the speedup analysis shown in Figure 8, we can make the following observations: smaller chunk sizes achieve the best speedup, closely following the ideal linear speedup trend.

Conversely, larger chunk sizes ($G = 128$) result in the worst speedup, as load imbalance and coarse task distribution limit parallel efficiency.

Static scheduling slightly outperforms dynamic scheduling, as it avoids task redistribution overhead. The farm implementation performs similarly to dynamic scheduling, indicating that stream parallelism does not provide significant improvements in this context.

Scalability

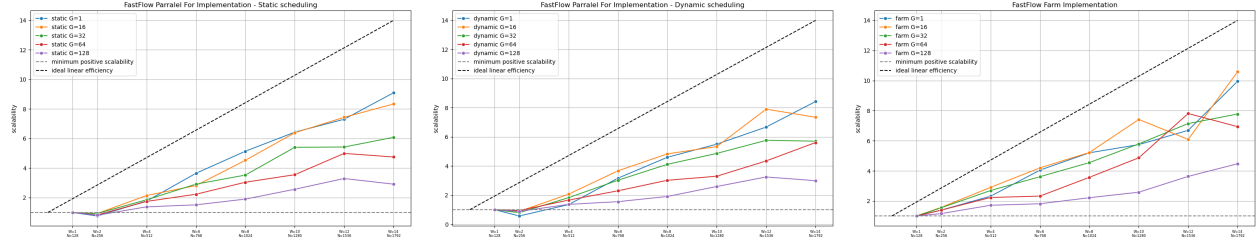


Figure 9: Scalability metrics for the weak scalability analysis of the **FastFlow** implementations. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

The scalability analysis, Figure 9, for parallel-for strategies confirms the observations made in the speedup analysis. The best scalability is achieved with smaller chunk sizes.

The farm implementation showing better scalability for small chunk sizes. Furthermore, unlike parallel-for, where $G = 128$ reaches a plateau after 12 workers, the farm implementation continues to scale beyond this point, suggesting that the task distribution overhead is handled more efficiently in this model.

These results suggest that the farm model is more effective at handling larger workloads, especially for higher granularity configurations, where it avoids the early saturation observed in the parallel-for approach.

Efficiency

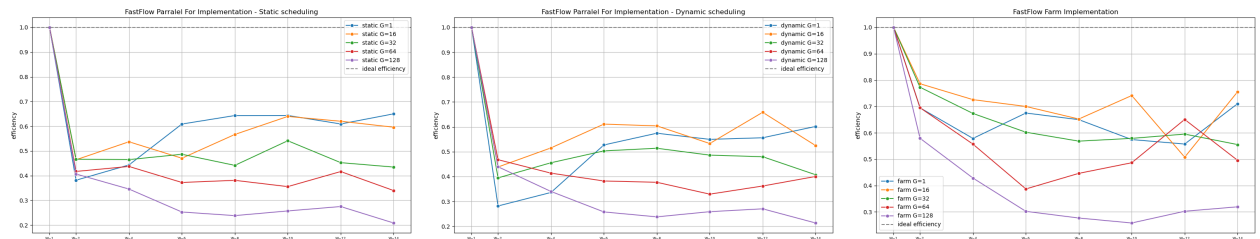


Figure 10: Efficiency metrics for the weak scalability analysis of the **FastFlow** implementations. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

As illustrated in Figure 10, parallel-for strategies exhibit relatively stable efficiency for $G = 1$ and $G = 16$, confirming that smaller chunk sizes optimize parallel efficiency. Larger chunk sizes show significantly lower efficiency, as load imbalance increases with coarse task distribution.

The farm implementation exhibits more fluctuations, with higher efficiency at small chunk sizes, but a noticeable drop for $G = 128$. However, efficiency does not degrade as quickly as in parallel-for, indicating that task distribution in the farm model mitigates some resource under-utilization. These results reinforce that fine-grained task distribution is crucial for maintaining efficiency, while larger granularity and synchronization overheads negatively impact resource utilization.

Cost

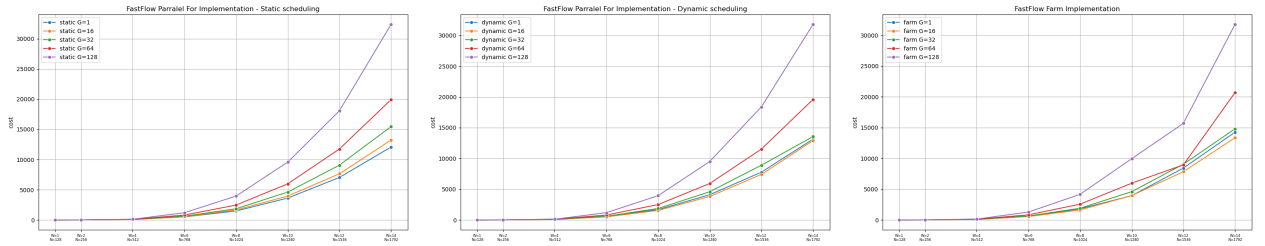


Figure 11: Cost metrics for the weak scalability analysis of the **FastFlow** implementations. The scaling started from a matrix size of 128 for a single thread and increased up to 14 workers, reaching a final matrix size of 3840. From left to right: parallel-for strategy with static scheduling, dynamic scheduling, and farm implementation.

For all implementations, cost increases with the number of workers, indicating diminishing efficiency as more resources are allocated, as we can see in figure 11. Test conducted with granularity $G = 128$ consistently results in the highest cost, confirming that larger chunk sizes lead to poor load balancing and inefficiencies. Lower granularity values ($G = 1, G = 16$) maintain lower costs, especially for smaller worker counts, indicating better resource utilization.

The farm implementation exhibits a cost trend similar to parallel-for but scales slightly worse for higher worker counts, suggesting that the overhead introduced by task distribution and synchronization in the emitter-worker model becomes more significant as the number of workers increases.

5 Performance Analysis: MultiProcess Implementation

For testing the multiprocess version, experiments were conducted on *spmcluster*, utilizing all 8 available nodes. Preliminary tests were performed to determine the optimal number of processes per node. The results indicated that assigning more than two processes per node degraded performance, likely due to the additional **OpenMP** threads required per process.

To evaluate the performance of the **MPI** implementation, experiments measured both strong and weak scalability, varying the number of processes and matrix size N .

5.1 Strong Scalability

The strong scalability tests were conducted with a fixed matrix size of 4096×4096 , varying the number of processes from 1 to 16.

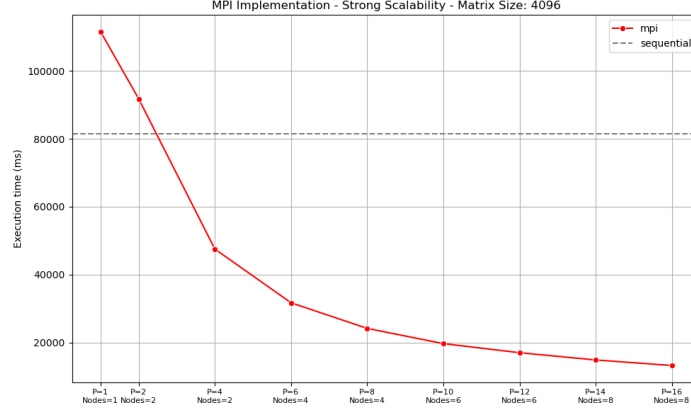


Figure 12: Execution time for strong scalability tests of the MPI implementation with a fixed matrix size of 4096×4096 , varying the number of processes from 1 to 16. The plot illustrates how execution time evolves as the number of processes increases.

As demonstrated in Figure 12, there is a substantial decrease in execution time as the number of processes increases, confirming that parallelization effectively reduces computation time. The most substantial performance improvement occurs when scaling to 4 processes, after which the reduction in execution time becomes more gradual. Beyond 8 processes, communication overhead becomes more significant, leading to diminishing returns in scalability.

In Figure 13, performance metrics for the strong scalability analysis are presented, leading to the following observations.

Both speedup and scalability increase linearly with the number of processes but deviate from ideal scaling, highlighting the impact of communication overhead and synchronization costs. This makes the choice of the number of processes crucial for optimizing performance.

Efficiency remains low even with just two processes and then declines gradually as more processes are added, reflecting diminishing returns due to inter-process communication overhead.

Notably, computational cost increases drastically between 1 and 2 processes, emphasizing the significant cost of MPI communication. As the number of processes grows, the cost continues to rise, confirming that resource utilization becomes less efficient at higher process counts due to increasing communication and synchronization overhead.

These results highlight that MPI parallelization effectively reduces execution time, but scalability is limited by communication overhead.

5.2 Weak Scalability

The evaluation of weak scalability was conducted by increasing the matrix size in proportion to the number of processes. The scaling process initiated with a matrix size of 512 for a single process and culminated in a matrix size of 8192, with 16 processes distributed across 8 nodes.

As evidenced by Figure 14, the execution time remains considerably lower than the sequential counterpart, yet does not remain constant. This validates the efficacy of multiprocess parallelisation in effectively handling increasing workloads and mitigating the growth in execution

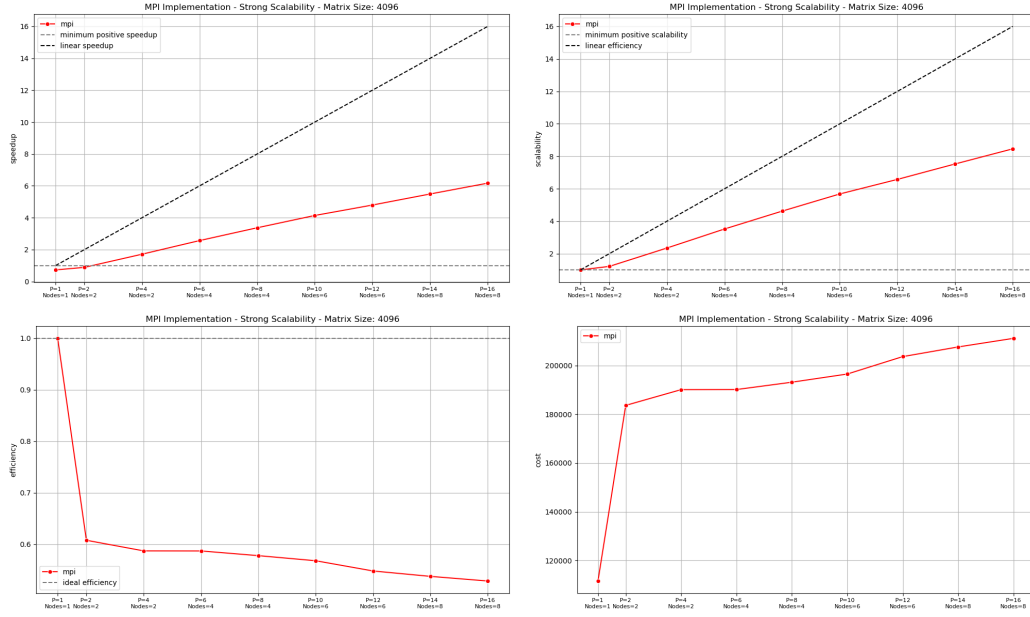


Figure 13: Performance metrics for strong scalability tests of the MPI implementation with a fixed matrix size of 4096×4096 , varying the number of processes from 1 to 16. The plots illustrate Speedup (top-left), Scalability (top-right), Efficiency (bottom-left), and Cost (bottom-right).

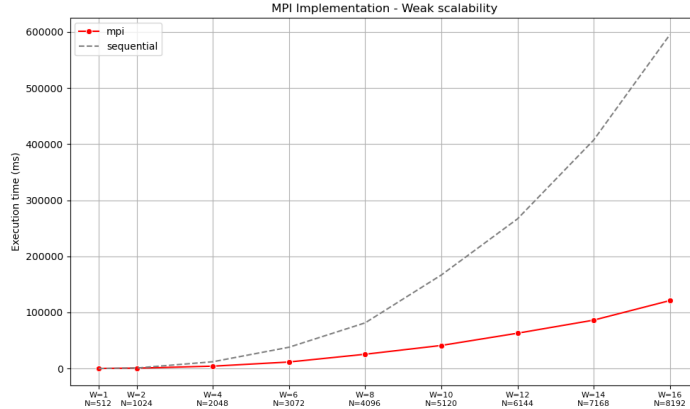


Figure 14: Execution time for weak scalability tests of the MPI implementation. The scaling process initiated with a matrix size of 512 for a single process and culminated in a matrix size of 8192, with 16 processes distributed across 8 nodes.

time as problem size increases. However, as communication overhead rises with an increase in processes, it results in a certain degree of performance degradation.

Figure 15 presents performance metrics for the weak scalability analysis.

Speedup and scalability increase as more processes are added, but deviate from the ideal linear trend, indicating the growing impact of communication and synchronization overhead. Up to 4 processes, and for smaller matrix sizes, speedup remains nearly linear, suggesting that parallelization is effective at this scale.

Efficiency declines steadily beyond 4 processes, reflecting the increasing parallelization overhead at larger scales.

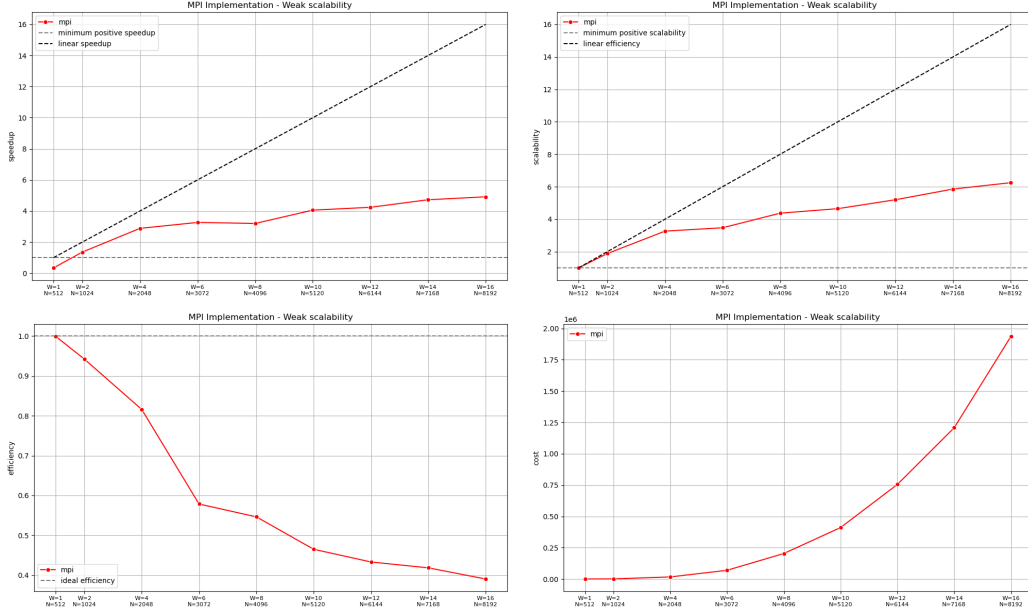


Figure 15: Performance metrics for weak scalability tests of the MPI implementation. The scaling process initiated with a matrix size of 512 for a single process and culminated in a matrix size of 8192, with 16 processes distributed across 8 nodes. The plots illustrate Speedup (top-left), Scalability (top-right), Efficiency (bottom-left), and Cost (bottom-right).

Additionally, computational cost grows exponentially, confirming that resource utilization becomes less efficient at higher process counts due to the increasing communication burden. This suggests that while MPI weak scalability is reasonable at lower scales, performance degradation becomes evident as the workload and number of processes grow.

6 Conclusions

This work analyzed different parallelization strategies for wavefront computation exploring both multithreads and multiprocess implementations.

The analysis of multithreads methods indicate that data parallelism (parallel-for) with static scheduling is the most efficient approach in strong scalability tests, as the computational cost per element within the same diagonal is constant. This ensures optimal load balancing with minimal scheduling overhead, making static scheduling particularly effective.

However, in weak scalability tests, the farm model shows better scalability at higher worker counts. This suggests that stream parallelism can mitigate some inefficiencies when scaling to larger problem sizes, likely due to its ability to dynamically distribute tasks as the workload increases.

For multi-process implementations, results showed good but sublinear strong scalability, with diminishing returns due to communication overhead. Efficiency remained low even with a small number of processes and continued to decrease as more processes were added. Weak scalability tests confirmed that MPI effectively mitigates execution time growth, but communication and synchronization costs become significant at larger scales, impacting performance.