Project 1: Distributed Wavefront Computation

Master Degree in Computer Science, University of Pisa
Parallel and Distributed Systems: Paradigms and Models
Academic Year 2023/2024

Giacomo Aru - g.aru@studenti.unipi.it - 597700

Abstract

This report explores different parallel strategies to optimize matrix wavefront computation, using both multithread and multiprocess approaches. Seven programs were implemented, each adopting a unique parallelism method, and were evaluated to assess their effectiveness in reducing execution time. This study outlines the methodology, theoretical background, key implementation details, and performance analysis of these parallel strategies.

1 WorkFlow

During the course of the project, an incremental implementation path was followed, adding complexity to the code by introducing parallelization and data management solutions to reduce execution time.

The first program to be implemented was `sequential_sm`, a sequential version of the wavefront computation. The program employs a simple data structure, specifically designed for the project, for the storage of the upper triangular matrix in a space-efficient manner. The structure uses a memory row-major ordering so that the rows of the matrix occupy contiguous memory locations. The logic of the program is implemented in a concise block of code, which serves as the program's core. This block of code calculates the values of each element of the matrix through the use of two nested `for` loops.

The first parallel implementation of the code was `fastflow_data_sm`, based on data-driven parallelisation using the `FastFlow` library. Upon examination of the sequential code, two direct parallelisation strategies emerged as potentially viable using the `ParallelForReduce` object. The first strategy entailed parallelising the dot product (multiplication element by element between the corresponding row and column vector, followed by a sum of the results), thereby computing each element in parallel. The second strategy involved parallelising the computation of the various elements in the current diagonal, which corresponded to a set of elements whose computations were mutually independent at each step. Subsequent to a series of tests, the second strategy was found to be more advantageous than the first.

At this stage of the project, a change in the handling and indexing of the data was considered. To calculate each matrix element, which involves a dot product, are needed many memory accesses, which can cause poor performance if they produce "cache misses". This can be avoided by exploiting spatial locality and trying to perform as many sequential accesses as possible. So it was implemented two new parallel versions of the previous programs: `sequential_dm` and `fastflow_data_dm`, which are "double matrix" versions of the previous "single matrix" (hence the names "dm" and "sm"). In these two new implementations, the data shared by threads is duplicated in a matrix stored in a column-major order.

Subsequently, two other parallel versions were implemented: `fastflow_stream_sm` and `fastflow_stream_dm`, which maintained the complementarity of data storage but changed the parallelization strategy. In these two programs, a `ff_Farm` was used to partition the problem into subtasks and assign them among the "workers" until the wavefront computation was completed.

Finally, the `mpi_wavefront` program was implemented, using OpenMPI to create and manage different processes. The computation of the final matrix was divided between the processes, similar to the last two multi-threaded programs. Each process is given a subtask to perform and shares the results with the other processes until the computation is complete. In addition, as in the first two multi-threaded programs, each process implements data-driven parallelisation via OpenMP to speed up the subtasks.

After the implementation of each program, various tests were performed to verify its correctness and to prepare it for a comparative study. These tests are not included in the report, but were designed to ensure that the programs worked as intended.

Makefiles and bash scripts were used to optimise the compilation and testing process. These tools made it possible to speed up the compilation of the programs by configuring the `DEBUG` value via the makefile, and to create benchmark tests to record execution times and compare results between different programs. Using these approaches, an in-depth analysis of the performance of the implemented programs was performed.

2 Parallelisation and Optimisation Strategy Adopted

In this section, we provide a more in-depth theoretical explanation of the approaches used, discussing the different options considered and the reasoning behind the decisions made.

2.1 Data structure

All the programs implemented can be grouped into two main categories based on the type of data structure they use. The first category is characterized by the use of a single instance of `TriangularMatrix` [1], a simple custom data structure that use an array to store space efficiently the matrix, in a row-major order. In this way is avoided redundancy of the data and useless waste of physical memory.

In the case of the single-process programs, the `TriangularMatrix` is shared among all processes that operate on it concurrently. To prevent race conditions, careful attention was given in the programming phase to ensuring thread-safe access, eliminating the need for explicit synchronization mechanisms such as "mutexes", which slow down the program and in this case, given the nature of the problem, are unnecessary.

In the single case of the multi-process program, was necessary for each process to store its own partial matrix, as the continuation of the computation needed the previously calculated elements. In fact, the problem exhibited strong recursive components, resulting in a temporal dependency between the calculation steps. Additionally, there was a spatial dependency among the elements, as each element depended on all those previously calculated in the same row and column.

However, all these implementations exhibited the same issue: a high number of cache misses during the calculation of element values. This occurred because the matrix values were stored in memory using row-major order, which led to "cache misses" whenever accessing elements in the iteration column.

To address this problem, it was decided to use a second copy of the matrix, this time stored in column-major order. By doing so, during the computation of the dot product, iteration over rows was performed using the primary "row matrix", while iteration over columns was handled by the secondary "column matrix". This approach sacrificed spatial efficiency, as the computational cost proved to be significantly higher than the space cost. Given this reason, it was decided not to employ temporary data manipulation techniques, as they would still impose a significant computational overhead.

The space required by the program comprises that necessary to store the entire matrix, in addition to the computational space, which can be reduced to a few memory locations, so it is $O(1)$. Therefore, the total space complexity is $O(N^2) + O(1) = O(N^2)$. In terms of computational cost, it is $O(N)$ to compute a single element, resulting in an overall cost of $O(N^2) \times O(N) = O(N^3)$ for the entire process.

The tests confirmed this observation, showing that, for every type of implementation, the single-matrix version was less efficient in terms of computational cost compared to the two-matrix version.

2.2 Multithread implementation, static approach

The implementation of `fastflow_data_sm` and `fastflow_data_dm` follow the first of two fundamentally different approaches employed. This approach is categorized as static data-parallelism, since the same operation is executed on multiple data elements by different computing units. Given the nature of the problem to be solved, this approach proves to be the only truly effective one, as dividing the process into tasks or different operations to be applied to the same data, and thus adopting a task-parallelism approach, is highly complex or unproductive.

The execution flow of the sequential program features three primary iteration loops, which could be subject to a brief yet effective modification to enable parallel execution. However, only two of these loops are actually suitable for this purpose:

[1] The `TriangularMatrix` class implementation was subsequent to the implementation of `UpperMatrix` class, which does not provide the capability to change the memory order. The primary difference between the two classes lies in the convenience of indexing offered by `TriangularMatrix`, which features automatic coordinate translation.

- The first loop involves iterating over all the upper diagonals of the matrix, which need to be computed recursively. Specifically, the diagonal at iteration `i` also depends on the elements of the diagonal computed from iteration `i - 1`. Therefore, this loop is not easily parallelisable, and it may never be worthwhile to attempt parallelisation here.
- The second loop involves iterating over all the elements of an upper diagonal that need to be computed. These iterations are mutually independent and produce values that need to be stored in the matrix, independently of the results of other iterations. Therefore, this section can be efficiently parallelised using a simple `ParallelFor`. This was indeed the solution adopted in the programs mentioned previously.
- The second loop involves iterating over all the elements in a row and a column of the matrix to compute a single element. While the iterations are mutually independent in terms of computation, the results must be accumulated into a single sum. To parallelise this section of the code, a `ParallelForReduce` can be used, with summation as the reduction operation. An initial implementation of this strategy was developed, but subsequent testing revealed that it was less effective than the previously mentioned approach. This inefficiency is likely due to the fact that the size of the arrays to be summed is proportional to the matrix size, $O(N)$. Since the computational cost remains $O(N^3)$, even relatively small array sizes can lead to excessively high workloads, so N will always be limited.

It is important to mention that the concurrent access to the data structure performed by threads in the approach described in point two, which is the one adopted, does not lead to race conditions. Although different threads require the same values from the computed partial matrix, access to memory locations occurs only in "read mode" once the element has been memorized. Given the "final" nature of the computed and stored value, there are no issues with this behavior.

2.3 Multithread Implementation, dynamic approach

During the course of the project, it was decided that two additional programs should be implemented for the purpose of testing the various capabilities of the `FastFlow` library, which is one of the focus of the project, as well as exploring a different parallelisation strategy. These was `fastflow_stream_sm` and `fastflow_stream_dm`. Although the new strategy also falls into the category of data parallelism, is dynamic and has a different logic and execution flow. This implementation uses a "farm" object and involves interactions between a source, workers and a sink.

To take full advantage from this methodology, a new strategy has been introduced that involves dividing the work into subtasks. Instead of considering the task as computing a single matrix element, it now entails a heavier workload, which justifies the overhead associated with the dynamic distribution of the computation. Indeed, the wavefront computation problem can be approached by simplifying it into a smaller, more manageable case, increasing the size of the subtasks. Specifically, the matrix can be partitioned into $M \times M$ blocks, where $M < N$. By focusing on this reduced $M \times M$ matrix, the problem becomes easier to handle. Within this smaller matrix, each block that appears in the upper diagonal can be computed independently, similar to the approach used in the original wavefront computation but applied to smaller upper diagonals.

In this scenario, the task is to compute all the elements within a single block. This computation can be performed by utilizing the values of elements from all previously computed blocks in the same row and column of the block itself. This new problem is also a generalized version of the wavefront computation that comprises all diagonals of the block, not just the upper ones, and can be tackled by understanding and applying the recursive formula inherent in the original computation. This new approach also allows for adjusting the workload of each task by changing the granularity of the subdivision for various purposes, such as optimizing cache usage of each threads.

It is important to note, however, that this new approach, while more complex and intriguing from an exploratory standpoint compared to the previous one, is theoretically unsuitable in certain respects. Although dynamic task distribution among workers appears to facilitate more effective resource management, it actually introduces a slowdown without providing any benefits, given that the sub-tasks are of equal computational complexity. The source distributes the subtasks to various workers almost always using a "round-robin" order, even in the absence of any imposed constraints. The use of unbounded buffer capacity for the worker, which undermines the dynamic nature of the program[2], leads

[2]To simulate on-demand task scheduling in the farm, it is possible to define the communication channels as bounded with a size of one, so that the source provides a new task only to those workers who need one.

to an improvement in performance, confirming that a more static approach yields benefits. However, the subdivision of the task enables a targeted approach to optimisation of the machine, thus facilitating greater efficiency in the code despite the presence of imperfections.

2.4 Multiprocess Implementation

The latest program developed for the project is `mpi_wavefront`. This program utilizes OpenMPI to implement a multi-process version of the wavefront computation, which previously had been computed using multi-threaded single-process programs. The logic behind this new implementation combines the two approaches mentioned earlier with an immediate use of two data structures.

Due to the inability to utilize shared memory because of the limitations imposed by the MPI paradigm [3] the decision was made to macroscopically use the first approach but with the work subdivision of the second approach. This combined approach aims to reduce overhead from inter-process communication by minimizing the number of iterations in which data is exchanged, at the cost of increasing the amount of data exchanged in each communication phase. Additionally, this approach allows for adjusting the granularity to optimize efficiency.

In addition, each process employs a multi-thread execution flow that follow more closely the first paradigm discussed earlier to solve each individual task. This approach allows for optimal utilization of the cluster's computing power available for testing the program.

Thus, the program flow involves statically assign to each process a contiguous block of tasks along the diagonal currently being computed. Each process then independently solves its assigned tasks using a second static division of the work among its threads. Finally, once each process has completed its assigned work, it broadcasts the blocks of computed elements to the other processes.

With a more in-depth study, a cleaner solution could potentially be developed to avoid this final global exchange of information, which entails a global data synchronization. This could involve taking advantage of the physical distribution of nodes and varying latencies in data exchange. However, due to the high spatial dependency of each element and, consequently, of each task, achieving this goal remains quite challenging and was not addressed as part of this project's work. Is also worth mentioning that the use of two data structures to efficiently index the matrix elements in the dot product does not affect the amount of data exchanged between processes. This is because the "duplication" of data occurs locally and is handled independently by each process.

3 Implementation Details

In this section, we provide a deeper exploration of the most critical implementation details of the code developed for this project, with a particular focus on the more complex aspects and those that directly illustrate the parallelisation strategy of the program.

3.1 Data structures

Designing an efficient data structure for storing and referencing the matrix elements was crucial to the success of the project. Although this challenge was not overly complex, it was addressed by leveraging two key elements detailed in Listing 1: the macro and the coordinate translation.

Listing 1: Code snippet from the `TriangularMatrix` data structure.

```
// Macro to calculate the index in the internal array for an element of coordinates (x,y).
#define UM_INDEXING(x,y) y + x * matrix_dimension - (x * x + x)/2

// Method to access an element at position (i, j)
// Handles column-major order if specified
matrix_type& element(int i, int j) {
    // Adjust indices if the matrix is in column-major order
    if (column_matrix) {
        int dummy_val = i;
        i = matrix_dimension - 1 - j;
        j = matrix_dimension - 1 - dummy_val;
    }
```

[3] limitations that are quite appropriate given that, in a multi-process environment, there is no guarantee of physical shared memory between the CPUs running the processes, except through explicit data exchange

```
13
14      // Return the matrix element at the computed index
15      return matrix_array[UM_INDEXING(i, j)];
16  }
```

The custom made `TriangularMatrix` data structure is capable of storing data in both row-major and column-major order depending on a constructor flag. However, to accurately reference the correct element in the memory array from the coordinates, it is essential to convert these coordinates precisely into an array index. To achieve this, the class first unifies the two different representations by adjusting the input coordinates when referencing a matrix stored in column-major order, and then calculates the correct array index using a macro proper to the data structure. The function returns a reference to the specified element of the array. Additionally, once this reference is obtained, it is possible to iterate directly over the memory.

3.2 Multithread implementation, static approach

The two programs, `fastflow_data_sm` and `fastflow_data_dm`, share a similar computation flow, differing only in their method of referencing memory elements when calculating new elements on the current diagonal. Both programs also adhere to the computation pattern of the sequential version as we can see in Listings 2 and 3.

After creating and initializing the matrices as specified in the project requirements:

> The values of the element on the major diagonal $e^0_{m,m}$ are initialized with the values $\frac{m+1}{n}$.

They iterate over all the upper diagonals other then the main diagonal to populate the matrix with values according to the ordering defined by the recursive relation:

$$e^k_{i,j} = \sqrt{\text{dotprod}(\text{vm}_k, \text{vm}^k_{+k})}$$

Each element is computed using the `parallel_for` function provided by the `FastFlow` library, which distribute the workload between n thread according to the specifications. In this case, a standard division for static distribution is used, where the i-th thread receives the i-th work block, with each block being a continuous segment of execution cycles. Therefore, thread i must complete the cycles with indices in the range $[i \cdot l, i \cdot (l+1))$. In each iteration, the element is computed by first performing a dot product between the corresponding row and column, and then the cubic root of the final sum is calculated. In the two "double matrix" programs, it is important to ensure that memory references are made through the correct matrix to minimize cache misses.

In order to implement the parallelisation is used the `parallel_for` class function of a previously created `ParallelForReduce` object, rather than the static parallelisation function also provided by the library, to optimize the program. This approach ensures that thread creation, along with its associated overhead, occurs only once, despite the execution of multiple parallel `for` loops.

3.3 Multithread implementation, dynamic approach

The implementation of `fastflow_stream_sm` and `fastflow_stream_dm` is among the most complex of the programs developed for this project, involving two custom classes interacting through a `ff_Farm` object, along with an innovative approach to divide work into subtasks.

In order to perform this decomposition, a novel parameter, designated as `granularity`, was introduced. This parameter controls the size of the subtasks and, consequently, their number. Additionally, an extra "presence" matrix was added to track the completion of subtasks by workers. For this purpose, an instance of `TriangularMatrix` with boolean values was employed, as only one bit of information is required for each subtask.

The first class part of the farm is `SourceSinkSingleMatrix`[4]. This class functions as both the source and sink of the farm: it handles the creation of new subtasks once all dependencies are resolved and sends them to the workers, and it also receives completed subtasks from the workers, updating the presence matrix to reflect their completion. The core functionality of this class is illustrated in Listing 4.

Indeed, a component was removed from the standard farm structure with the objective of unifying the source and sink roles. It was determined that a separate sink component was unnecessary, as each thread was capable of independently storing computed results in the matrices. Also, updating the

[4] `SourceSinkDoubleMatrix` is the "double matrix" counterpart

presence matrix to notify subtask completion needed to occur concurrently with the creation of new subtasks, which could lead to race conditions if not managed correctly. To avoid delays from explicit synchronization mechanisms, such as "mutexes", this responsibility was assigned to the source class. By handling both the creation of new subtasks and updating the presence matrix, the source class effectively prevents concurrent conflicts. Moreover, updating the presence matrix is not particularly resource-intensive.

The second class, named `WorkerSingleMatrix`[5], implements the workers responsible for performing all computations necessary to derive the values within each block, which corresponds to a single subtask. It is important to note that, although multiple workers share the same data structure, there are no race conditions. This is because all operations on elements of completed subtasks are read-only and involve no writing. Furthermore, only the worker assigned to a specific subtask needs access to the elements that are part of that subtask itself. This setup reflects the original wavefront computation problem, and by carefully managing the order of calculations, race conditions are avoided.

Listing 5 illustrates the two phases of computing each subtask: the first phase involves calculating all the elements in the lower diagonal of the block, while the second phase handles the remaining elements, including the main diagonal. In both phases, the computation progresses from the lowest diagonal, which includes the bottom-left element, to the highest diagonal, which includes the top-right element. Note that the execution of the first block of code only occurs if we are not in the initial iteration of the wavefront computation, as the first iteration begins with a diagonal that has already been initialized. Thus, after the first iteration, the macroscopic block structure of the original element matrix is achieved, and the subtasks are more distinguishable.

3.4 Multiprocess Implementation

The implementation of `mpi_wavefront` combines the key features of the previously described programs, such as the decomposition into subtasks with the associated management of granularity, and the static distribution of work among various computation streams, achieved through dividing the computation of elements along the diagonal. Additionally, it introduces significant changes not only to these two strategies but also incorporates the most notable difference: the adoption of a multi-process architecture implemented using MPI (Message Passing Interface).

The division into subproblems follows the logic implemented in `fastflow_stream_sm` but since the workload distribution follows the logic implemented in `fastflow_data_sm`, the subtasks are statically, and not dynamically, distributed among the active processes, so the presence matrix is not needed since the workload distribution itself induces the correct computation order of matrix values, preventing race condition in the writing and reading operations.

However, each process must exclusively operate on its own memory, which remains inaccessible to others. Upon completing the computation, a global data exchange phase is necessary for all processes to memorize the computed elements and continue the computation. This is achieved using the OpenMPI function `MPI_Allgatherv`, which allows information exchange through message passing: each process sends its portion of the array to the other processes and receives portions computed by them.

The function `MPI_Allgatherv` is used instead of `MPI_Allgather` because not all processes necessarily compute the same number of elements. This can happen for two main reasons:

- The global computation is approaching completion, with only k tasks remaining to be computed in parallel, where since k begins as $\frac{N}{M}$ and decreases to 1 at some point when $k < P$ (the number of processes) there are more idle processes than pending task.
- M does not perfectly divide N, causing the last process to handle a smaller-sized task compared to the others.

Finally, each process performs a parallel computation of its own task. As shown in Listing 6, an OpenMP (OMP) directive is used to parallelise both phases of the task computation. However, the parallelisation is limited to the computation phase and does not extend to the subsequent memorization phase. In fact, after conducting several tests, it was observed that, given the nature of the memorization phase, that primarily involve memory access operations, parallelisation only leads to congestion in data transfer between caches.

[5] `WorkerDoubleMatrix` is the "double matrix" counterpart

4 Performance Analysis

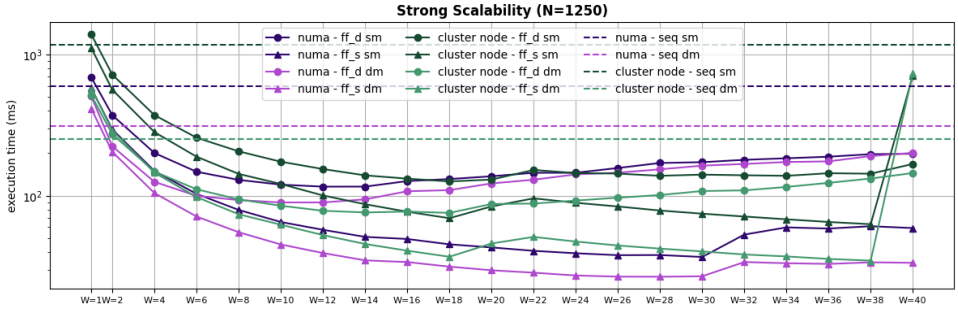
This section presents the results of the performance tests conducted on all programs, divided into three parts. The first two sections examine the strong and weak scalability of multithreaded programs in relation to their sequential counterparts. The latter section provides a corresponding analysis for the multiprocess program.

For all tests, each collected value is the result of a statistical average derived from multiple executions (bethween three and six, depending on the availability of the shared machine and workload) of the same parameter combination. This approach is used to avoid inaccuracies caused by temporary performance drops.

4.1 Multithread Strong Scalability

To accurately compare the various programs, especially those utilizing dynamic scheduling, it is crucial to properly set the granularity parameter. This ensures that the analysis focuses solely on variations in problem size N and the number of threads W. Preliminary tests were conducted with the `fastflow_stream_sm` and `fastflow_stream_dm` programs to determine an optimal granularity value, which is machine-optimized and independent of the number of threads. The results of these tests are detailed in Appendix A. Based on these results, a granularity value of 16 was chosen for the machine at address "spmnuma.unipi.it" (referred to as "numa" in the plots), and a value of 8 was selected for a cluster node at address "spmcluster.unipi.it" (referred to as "cluster" in the plots).

Figure 1: Strong scalability analysis for all multithreaded programs. The x-axis shows the number of threads and the y-axis the execution time in milliseconds. The matrix size is fixed at 1250.



The strong scalability analysis is presented in Figure 1. This figure includes all single-process programs and their sequential counterparts. The x-axis represents the number of threads used, while N is fixed at 1250. The y-axis shows the execution time in milliseconds on a logarithmic scale to enhance readability and distinguish the curves. A linear scale version is provided in Figure 13 for reference.

Dashed lines are references to the sequential versions of the parallelised algorithms, represented by the continuous lines. Notably, the sequential double matrix versions significantly outperform the single matrix versions, with differences even by almost an order of magnitude (1150 ms vs. approximately 250 ms) in the cluster scenario. This pattern persists among the parallel programs, although the gap varies with the number of threads. This demonstrates how sequential access to memory can drastically affect execution times compared to random access.

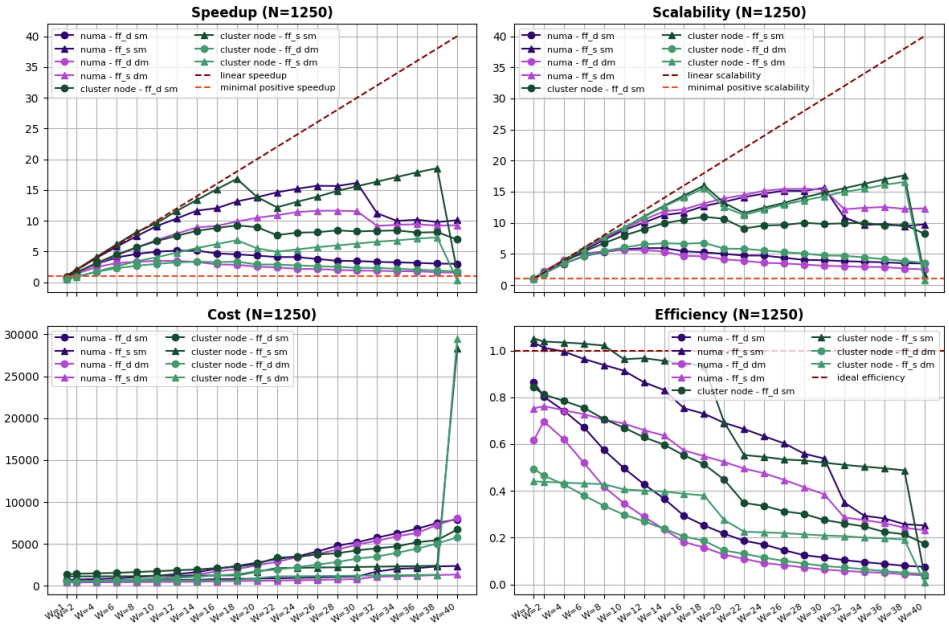
At very low thread counts, all versions perform worse than their sequential counterparts. The most significant difference is observed in `fastflow_stream_dm` with $W = 1$, though this program performs among the best as the number of threads increases. The exception is `fastflow_stream_sm`, which performs slightly better than the sequential version.

These differences can be attributed to the influence of the execution machine's specific software and hardware characteristics, including memory cache distribution, size, structure, and the physical distance between cores. The graph clearly shows that the same code executed on different machines ("numa" and "cluster") yields vastly different performance results. In fact, the "numa" machine exhibits both the best and worst performances depending on the program.

The graph also reveals significant performance drops between consecutive points, particularly between $W = 18$ and $W = 22$, $W = 30$ and $W = 32$, and especially between $W = 38$ and $W = 40$ for `fastflow_stream` programs on a cluster node. This is due to the number of cores in each node (40 cores). When $W = 40$, there are actually 41 execution flows, including the source-sink of the farm,

exceeding the number of available cores, which can lead to excessive "context switching" detrimental to performance, especially for memory-bound programs.

Figure 2: Speedup, scalability, cost and efficiency calculated with reference to the data shown in Figure 1



The "stream" program versions generally outperform the "data" versions. For example, the worst-performing program in the "stream" category (`fastflow_stream_sm` executed on a cluster) consistently surpasses the best-performing program in the "data" category (`fastflow_data_dm` executed on a cluster), except for a few points on the graph where this trend does not hold.

Finally, efficiency metrics show a good reduction in execution time initially, followed by an increase in the second half of the graph. This shift occurs around $W = 12$ for some programs and around $W = 30$ for others. The performance metrics graphs in Figure 2 provide a clearer view of the parallelisation performance compared to sequential versions. Four metrics are shown for each curve: speedup, efficiency, scalability, and cost, with dashed lines indicating references to minimal or optimal behaviors.

The cost of each program tends to increase with an extreme rise in the previously mentioned case, where there is a drastic drop in performance for `fastflow_stream` programs. Therefore, there is an increase in cost for parallel execution across all versions. This is also confirmed by the consistent decrease in efficiency of each program. The "stream" program versions are the most efficient, as they maintain an acceptable efficiency for most of the graph, although they also tend to decrease significantly towards the end. Speedup and scalability show similar curves due to their nature, but scalability is more lenient with each parallel version, as parallelisation introduces overhead compared to the sequential version, though this is justified by the potential to speed up computations.

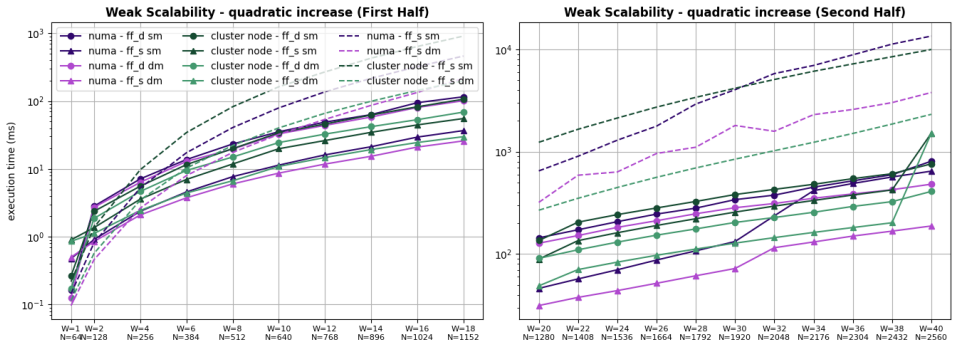
Finally, programs whose sequential versions were more performant have still shown good results in speedup, reaching high values in the first part of the plot. This behavior is particularly pronounced in the scalability plot, which demonstrates the ability of these programs to perform well with the increase in the number of threads. Finally, it is important to remember that these graphs should always be considered in relation to the actual computation time, as they show relative rather than absolute metrics. Indeed, a good speedup and efficiency do not necessarily make a program the best, since, as in these cases, it is possible to significantly improve the sequential version, and thus the reference point may not be absolutely reliable.

4.2 Multithread Weak Scalability

A weak scalability analysis entails measuring the computational time as the number of threads and problem size increase. This approach is less stringent than the strong scalability counterpart.

In this case, the problem size depends on the size of the matrix; however, the actual number of tasks is represented by the number of matrix elements, which is quadratic with respect to the side length. One

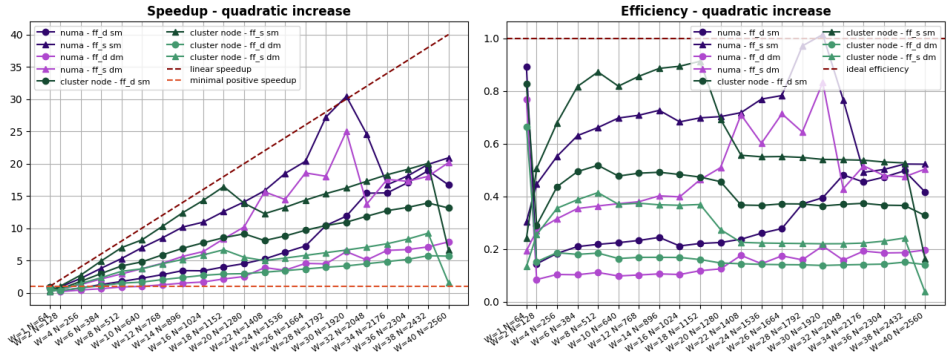
Figure 3: Weak scalability analysis for all multithreaded programs. The x-axis shows the number of threads and the y-axis the execution time in milliseconds. The matrix size increases by 64 for each additional thread.



could argue that a proper weak scalability analysis should involve a linear increase in the number of tasks and a sub-linear increase in the matrix side length. However, since the problem has strong dependencies among the tasks, it has only been possible to parallelise it in each computation iteration (computation of the current diagonal). Therefore, we will proceed with a quadratic increase for each step of the number of tasks, so that the parallelised section is subjected to a linear increase. Graphs representing an analysis with respect to a linear increase are available in Appendix C.

The graph shown in Figure 3 is the counterpart of the one present in Figure 1, but for weak scalability analysis. The only difference is the division of the x-axis. This modification was implemented to maximize the distance between the curves, thereby enhancing the overall readability of the graph. The two graphs represent the same quantity, but over different intervals of thread count.

Figure 4: Speedup and efficiency calculated with reference to the data shown in Figure 3



In the graph, we observe a curve ordering from top to bottom that follows the trend commented before, with the exception of a brief anomaly for the program `fastflow_stream_sm` executed on cluster node at the end. This definitively demonstrates that some implemented techniques are more effective than others. The program `fastflow_stream_dm` proves once again to be the most efficient, while `fastflow_data_sm` is the least efficient.

The difference in execution times between the sequential and parallel versions is significant, highlighting the importance of parallelisation. Moreover, each curve in the graph is monotonically increasing, indicating that with each increase in the number of threads, there is a performance gain, albeit sometimes minimal, and an increase in speedup as shown in the Figure 4[6]. Additionally, we observe an intriguing case of superlinear speedup at point W=4 for the program `fastflow_stream_sm`, due to a substantial performance gain from excellent data caching.

Apart from this particular case, many other curves show good speedup, especially in "stream" programs compared to "data" ones, and in "double matrix" programs compared to "single matrix" ones. Excluding the last point on the graph due to the usual cluster node anomaly, despite the excellent per-

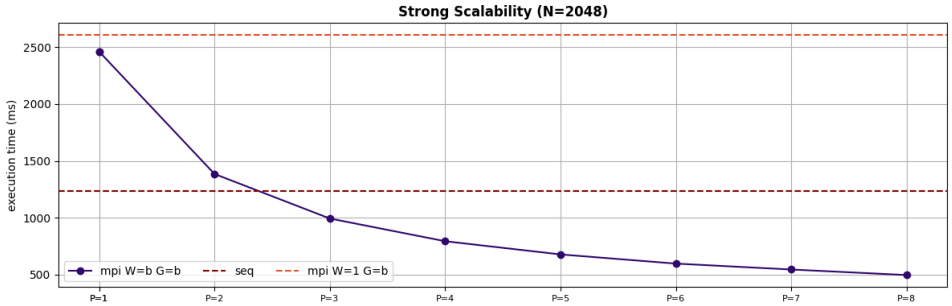
[6]The graph includes only speedup and efficiency, as calculating scalability would have required very long tests to measure the execution time of each parallel program with only one worker

formance achieved by the double matrix programs which remain the most efficient, the problem size reached in the final points of the graph (where N = 2432, resulting in a matrix with 5,914,624 elements!), could lead to memory issues. Therefore, it is always important to consider the trade-off made to achieve such efficiency gains when N grows significantly. Even though, given the quadratic increase, a linear reduction (halving the data) achievable with the "single matrix" version might still not be sufficient.

4.3 Multiprocess Scalability Analysis

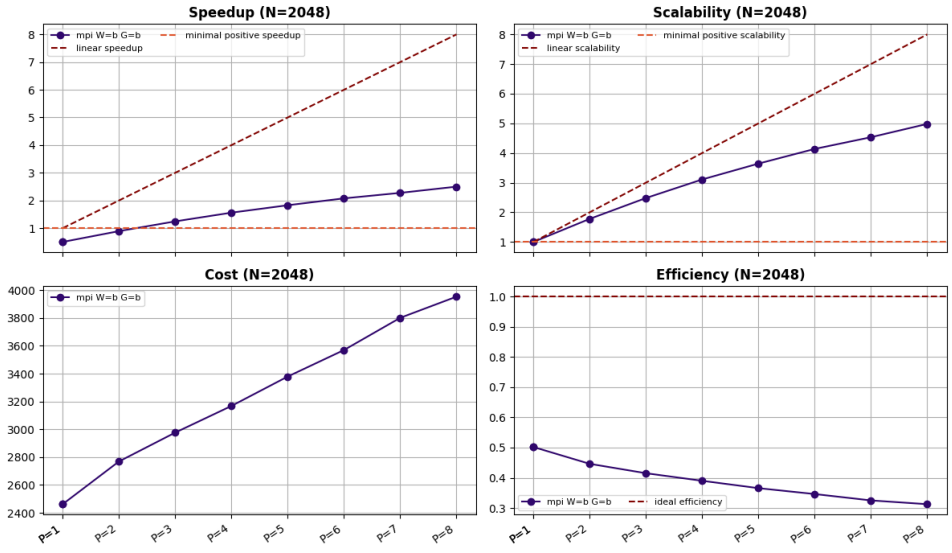
Once more, prior to undertaking the parallelisation analysis, preliminary tests were conducted with the objective of establishing the optimal granularity parameter value. In this instance, the values selected were one per different number of processes tested, given that the single machine and limited number of performed tests permitted enhanced precision in the selection. The results of this analysis are presented in Appendix A, accompanied by a brief explanation of the methodology employed.

Figure 5: Strong scalability analysis for the multiprocess program. The x-axis shows the number of processes and the y-axis the execution time in milliseconds. The matrix size is fixed at 2048. In the legend, the "b" value means the best possible value.



The principal outcome of this preliminary investigation was that, despite the "double" parallelisation approach dividing the calculation into processes and threads, the tests indicated that, for any number of processes and any size of table, at most two threads per process and never more was the optimal number. It would appear that the benefit derived from this approach was not greater than the loss incurred in parallelisation, which was probably due to the high number of memory reads and writes performed by the program, now that there is also an additional data-sharing phase.

Figure 6: Speedup and efficiency calculated with reference to the data shown in Figure 5. In the legend, the "b" value means the best possible value.



The results of the strong scalability analysis are presented in Figure 5. Additionally, the figures include references to the computational time of the sequential version and the multiprocess version utilising a

single process and a single worker. Evidently, an increase in the number of processes involved results in a notable performance gain, although it is less impactful than that observed in the multithreaded version.

A similar consideration can be done by examining the graphs of the calculated metrics in Figure 6, which illustrates the relationship between speedup and scalability. It is notable that, at each point, speedup is less than scalability. Additionally, as execution time decreases, cost and efficiency tend to increase, while the number of computation flows rises at a faster rate.

Moreover, the growth in speedup is not linear but rather sub-linear. This behaviour indicates an increasing inefficiency of the code, which exhibits a diminished capacity to improve performance as the number of processes increases. This phenomenon can be attributed to the necessity of a data sharing phase inherent to the problem, which necessitates a considerable amount of time to update each process element matrices.

Figure 7: Weak scalability analysis for the multiprocess program. The x-axis shows the number of threads and the y-axis the execution time in milliseconds. The matrix size increases by 512 for each additional thread. In the legend, the "b" value means the best possible value.

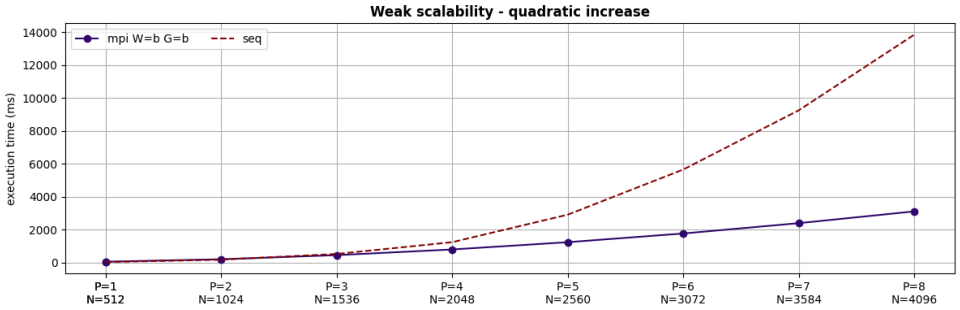
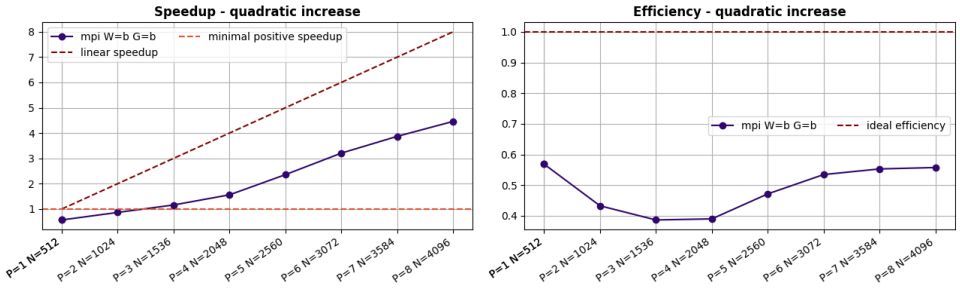


Figure 8: Speedup and efficiency calculated with reference to the data shown in Figure 7. In the legend, the "b" value means the best possible value.



The program responds better to a weak scalability analysis, as the curves represent a higher performance gain. In fact, the difference between sequential and parallel execution time in the graph in Figure 7 is remarkable. Also shown in the graphs in Figure 8 is a very good speedup, which towards the end again shows a tendency to become sublinear, but the previous maximum 2.5 speedup, achieved with 8 processes is now beaten by a much higher 4.5, and a cost that does not drop as the number of processes increases. In fact, we note that the efficiency shows an initial drop followed by a good increase.

A Granularity Tests

The graphs presented here describe the study on granularity conducted for `fastflow_stream_sm`, `fastflow_stream_dm` and `mpi_wavefront`. The goal was to identify optimal granularity values in order to analyze the programs' behavior as other parameters change. Modifying the granularity affects the size of the subtasks, which are distributed among execution flows, threads, or programs. A granularity of k divides the $N \times N$ matrix into M $k \times k$ blocks. If k is not a divisor of N, the last subtask will be smaller, and the program will handle the exception correctly.

Figure 9 shows the study for program `mpi_wavefront`, where three different matrix sizes were tested. The values represented are the logarithm of execution time in microseconds.

Figure 9: Heatmap of the logaritmic execution time of `mpi_wavefront`

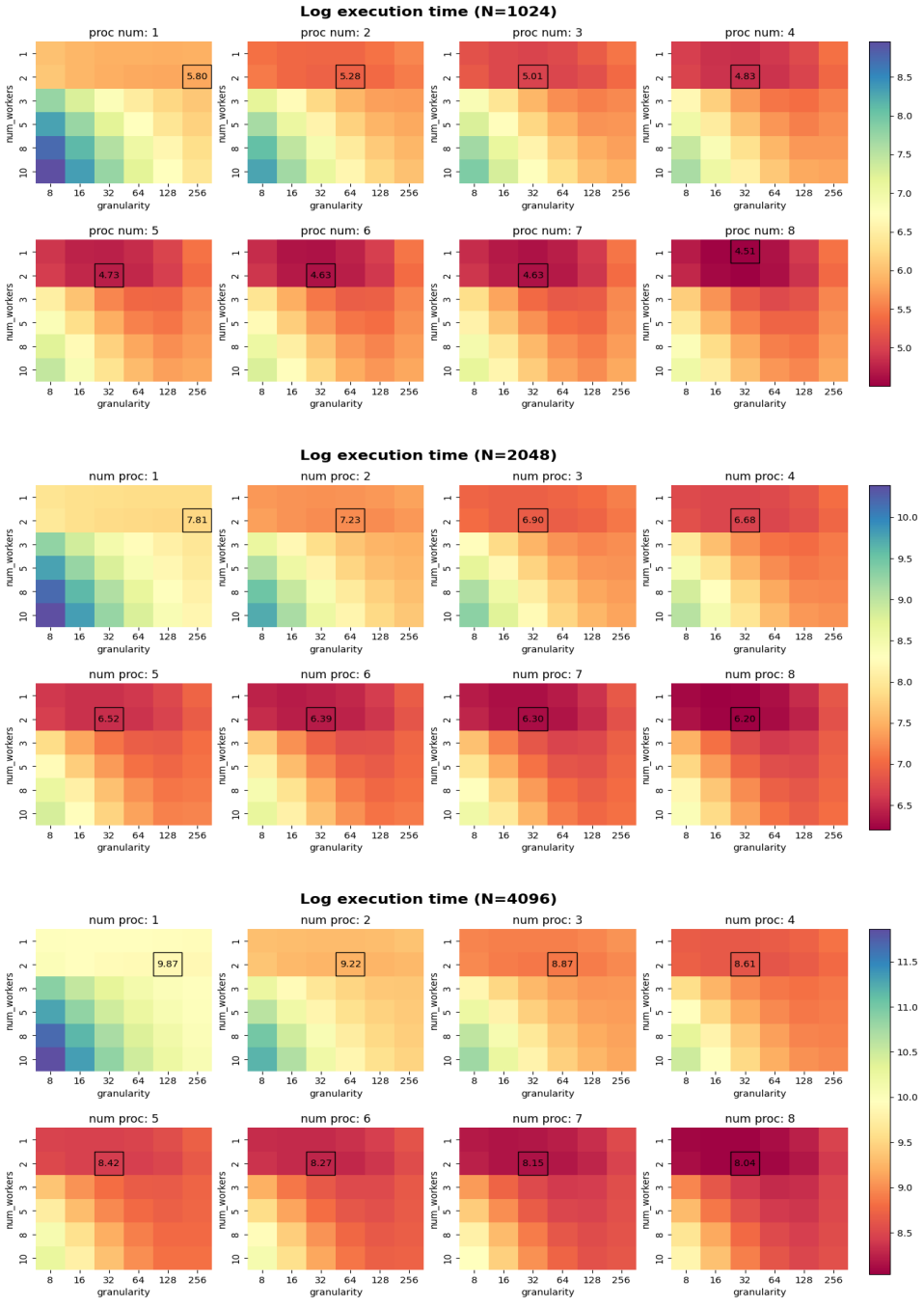
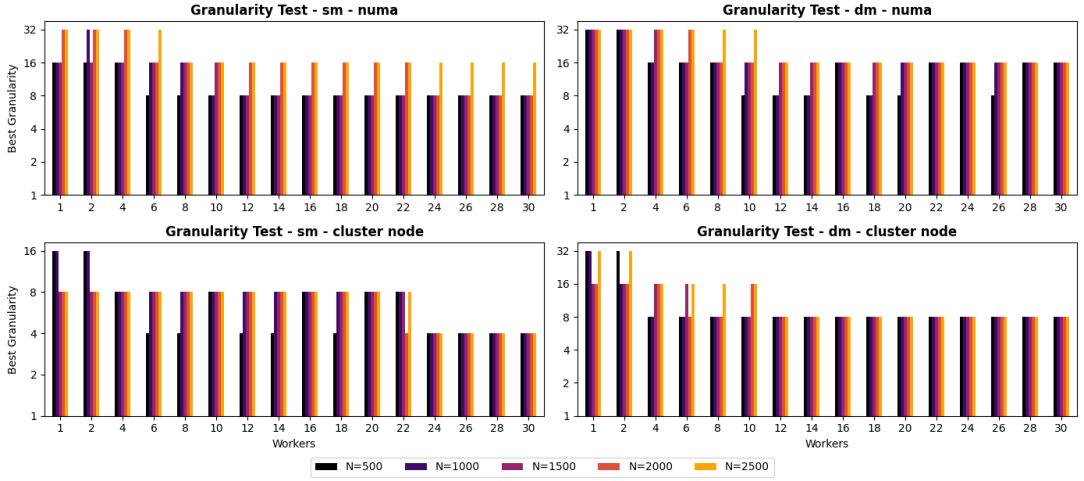


Figure 10 presents the study for program B, where five different matrix sizes were analyzed. For each

test machine, a different optimal value was identified, as the best granularity is significantly influenced by the hardware structure and memory size of the machine.

Figure 10: For each combination of N and W, is shown the best granularity, defined as the one that achieved the minimum execution time.



B Additional Code

Listing 2: Core loop of `sequential_sm` that implements the wavefront computation; Here, we can see the use of `DEBUG_CHECK` to intelligently print debug information.

```
// Perform the wavefront computation for each diagonal
for (int diag = 1; diag < um.matrix_dimension; diag++){
    // Compute a value for each element of the diagonal
    for (int k = 0; k < (um.matrix_dimension - diag); k++){
        double partial_sum = 0.0;

        for (int i = 0; i < diag; i++){
            partial_sum += um.element(k, k + i) * um.element(diag + k - i, diag + k);
        }

        // Update matrix element with the cube root of the partial sum
        partial_sum = std::cbrt(partial_sum);
        um.element(k, diag + k) = partial_sum;
    }
}
```

Listing 3: Core loop of `fastflow_data_sm`; Here we can see the use of `parallel for` to divide the work among the threads.

```
// Perform the wavefront computation for each diagonal in parallel
// The work is divided equally for each thread dividing the diagonal at each external iteration
for (int diag = 1; diag < um.matrix_dimension; diag++) {
    long v = (um.matrix_dimension - diag);
    // There's no need to use a rediuction operator as we're going to write the results in memory.
    // Also, the results on each diagonal are independent of each other.
    pf.parallel_for(0L, v, 1L,
                    0L,
                    [&](const long k) {
                        double partial_sum = 0.0;

                        for (int l = 0; l < diag; l++) {
                            // Compute partial sums using elements from the matrix
                            partial_sum += um.element(k, k + l) * um.element(diag + k - l, diag + k);
                        }

                        // Update matrix elements with the cube root of the computed partial sum
                        partial_sum = std::cbrt(partial_sum);
                        um.element(k, diag + k) = partial_sum;
                    });
}
```

Listing 4: Core logic of the source of the farm implemented in `fastflow_stream_dm`

```
// tuple unpacking
int i = std::get<0>(*task);
int j = std::get<1>(*task);

// Mark the received task as processed
presence_um.element(i,j) = 1;

// tasks are sent if they can be calculated, depending on the presence of the two adjacent tasks.
if (i-1 >= 0  && j-1 >= 0 && presence_um.element(i-1, j-1))
    ff_send_out(new task_source(&row_tm, &col_tm, i-1 , j, granularity, 0));
if (i+1 < pres_matrix_dimension  && j+1 < pres_matrix_dimension && presence_um.element(i+1, j+1))
    ff_send_out(new task_source(&row_tm, &col_tm, i , j+1, granularity, 0));
```

Listing 5: Core logic of the workers of the farm implemented in `fastflow_stream_dm`; Here, we can observe the use of the two different matrices in the computation, as well as the calculation of all the elements within each block (task).

```
// If this is NOT the first iteration, calculate the lower triangular part of the task,
// if this is the firts iteration, the lower triangular matrix is not needed nor computable
if (!first_it) {
    // Iterate over diagonal
    for (int diag = 0; diag < granularity - 1; diag++){
        // Iterate over diagonal element
        for (int k=diag; k>=0; k--){
            // Compute the absolute (element matrix) coordinates
            abs_i = i*granularity + (granularity - k - 1);
            abs_j = j*granularity + 1 + (diag - k);

            // Break if out of bounds
            if (abs_i >= row_tm->matrix_dimension || abs_j >= row_tm->matrix_dimension)
                break;

            // Update element matrices with the computed value
            dummy_val = compute(row_tm, col_tm, abs_i, abs_j);
            row_tm->element(abs_i, abs_j) = dummy_val;
            col_tm->element(abs_i, abs_j) = dummy_val;
        }
    }
}

// Compute upper triangular section of the task for all iteration
// (this use the lower part if is not the first iteration)
// Iterate over diagonal
for (int diag = 0; diag < granularity; diag++){
    // Iterate over diagonal element
    for (int k=0; k< (granularity - diag); k++){
        // Compute the absolute (element matrix) coordinates
        abs_i = i*granularity + (k);
        abs_j = j*granularity + 1 + (diag + k);

        // Break if out of bounds
        if (abs_i >= row_tm->matrix_dimension || abs_j >= row_tm->matrix_dimension)
            break;

        // Update element matrices with the computed value
        dummy_val = compute(row_tm, col_tm, abs_i, abs_j);
        row_tm->element(abs_i, abs_j) = dummy_val;
        col_tm->element(abs_i, abs_j) = dummy_val;
    }
}
```

Listing 6: Core logic of `mpi_wavefront`

```
// Each process compute the respective section of the task matrix diagonal
for (int k = proc_start; k < proc_end; k++){

    /* Each task is a similar subproblem to the original one and is solved in a similar way,
       in which case the lower triangular matrix must also be taken into account
       for all tasks that are not in the first daigonal*/
    int i = k, j = diag+k;
    int stop;

    // skipped in the first iteration, lower triangular part of the subtask
    if (diag != 0) {
        // iterate over the element of the diagonal
        for (int internal_diag = 0; internal_diag < granularity - 1; internal_diag++){

            stop = std::max(-1,
                        std::max(i*granularity + granularity - 1 - matrix_size,
                        j*granularity + 1 + internal_diag - matrix_size));

            // OpenMP is used to easily parallelise this cycle,
            // in which each iteration is independent of the others.
            #pragma omp parallel for
            for (int internal_k=internal_diag; internal_k>stop; internal_k--){

                // Here, the coordinates relative to the element matrix are calculated
```

```
                    and used
25                  int abs_i = i*granularity + (granularity - internal_k - 1);
26                  int abs_j = j*granularity + 1 + (internal_diag - internal_k);
27
28                  // The send buffer is popolated
29                  int atual_buff_it = buff_it + internal_diag - internal_k;
30                  sendbuf[atual_buff_it] = compute(&row_tm, &col_tm, abs_i, abs_j);
31                  row_tm.element(abs_i, abs_j) = sendbuf[atual_buff_it];
32                  col_tm.element(abs_i, abs_j) = sendbuf[atual_buff_it];
33              }
34              // The send buffer iterator is increased
35              buff_it += internal_diag - stop;
36          }
37      }
38      // upper triangular part of the subtask
39      for (int internal_diag = 0; internal_diag < granularity; internal_diag++){
40
41          stop = std::min(granularity - internal_diag,
42                          std::min(matrix_size - 1 - internal_diag - j*granularity,
43                          matrix_size - i*granularity));
44
45          // OpenMP is used to easily parallelise this cycle,
46          // in which each iteration is independent of the others.
47          #pragma omp parallel for
48          for (int internal_k=0; internal_k < stop; internal_k++){
49
50              // Here, the coordinates relative to the element matrix are calculated and
                    used
51              int abs_i = i*granularity + (internal_k);
52              int abs_j = j*granularity + 1 + (internal_diag + internal_k);
53
54              int atual_buff_it = buff_it + internal_k;
55
56              // The send buffer is popolated
57              sendbuf[atual_buff_it] = compute(&row_tm, &col_tm, abs_i, abs_j);
58              row_tm.element(abs_i, abs_j) = sendbuf[atual_buff_it];
59              col_tm.element(abs_i, abs_j) = sendbuf[atual_buff_it];
60          }
61          // The send buffer iterator is increased
62          buff_it += stop;
63      }
64  }
65
66  /*Here, the function MPI_Allgatherv is used to enable the exchange of computed elements
        between processes
67  and ensure that each process has all the elements calculated up to this point.
68  Each subsequent iteration may require the presence of these elements.*/
69  // Address of receive buffer (choice)
70  double* recvbuf = new double[total_work_load*task_work_load];
71  MPI_Allgatherv(sendbuf, real_proc_work_load * task_work_load , MPI_DOUBLE,
72                  recvbuf, recvcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
```

C Additional Plots

Figure 11: thread linear increase weak scalability analysis

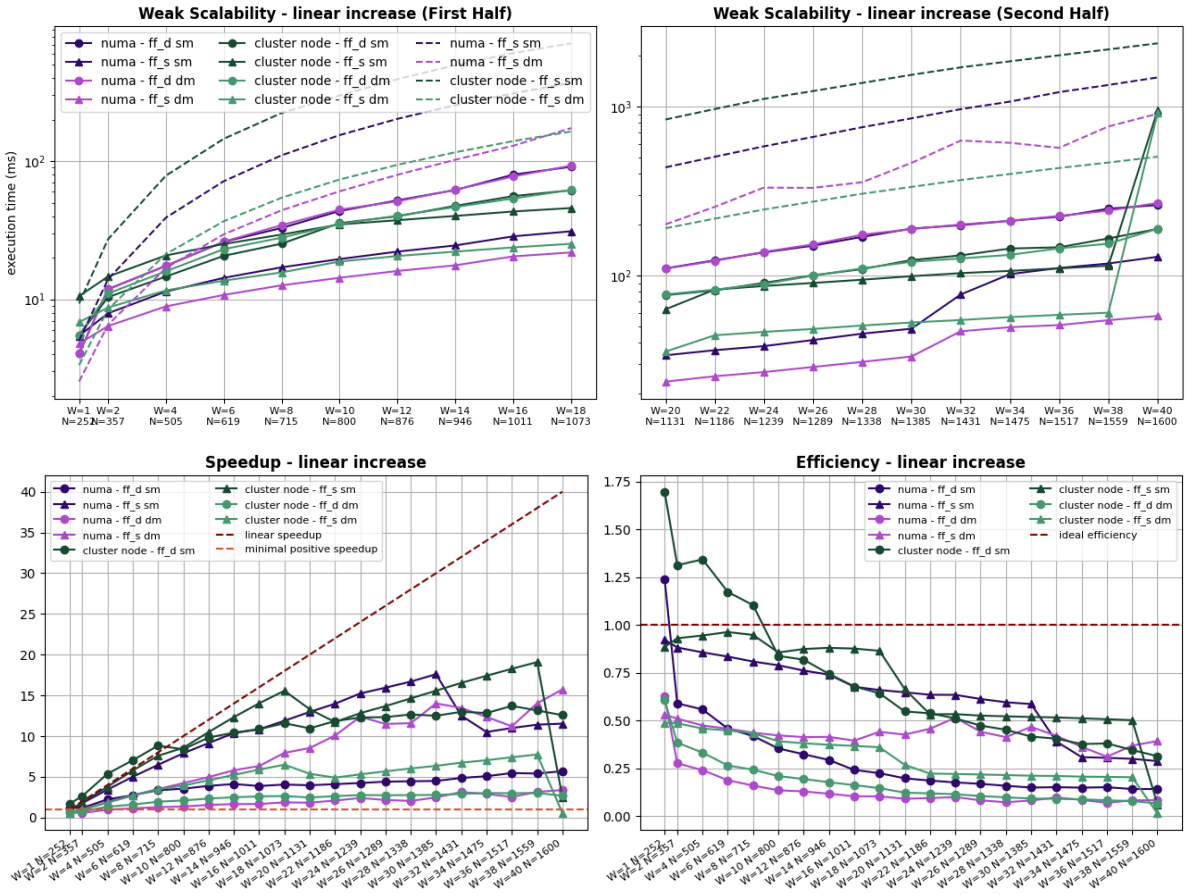


Figure 12: thread strong scalability, with linear scale

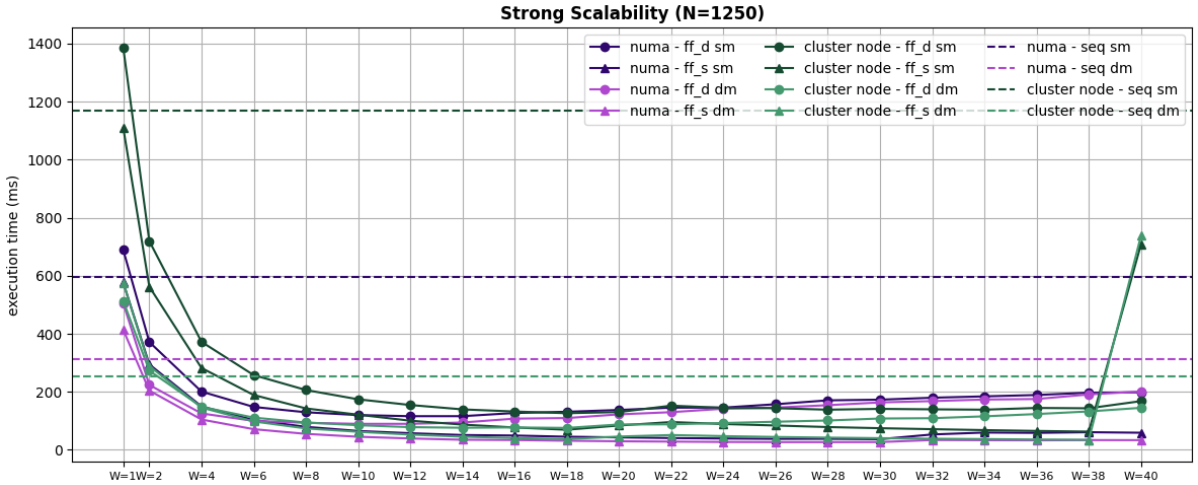


Figure 13: thread quadratic increase weak scalability, with linear scale

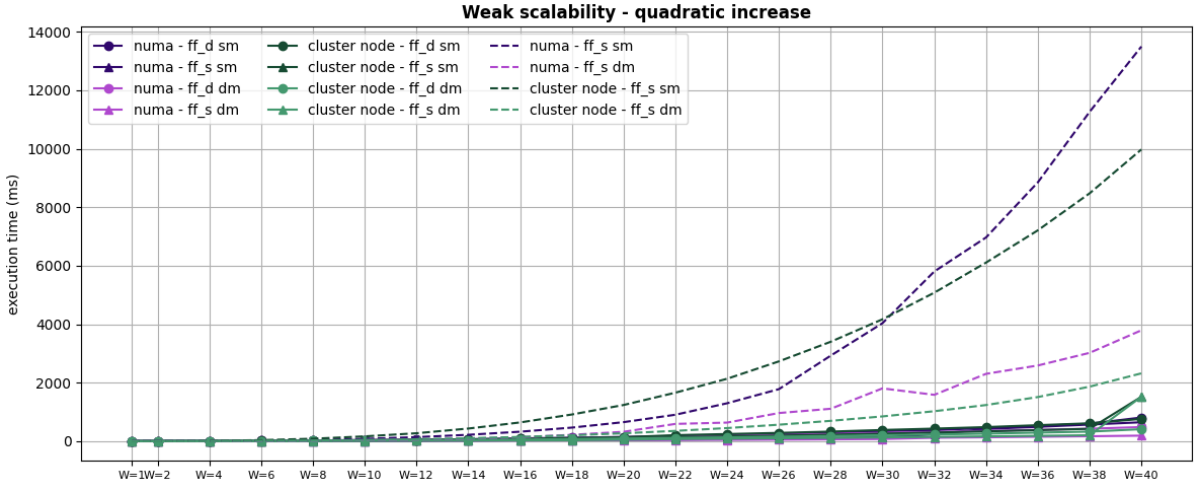


Figure 14: thread linear increase weak scalability, with linear scale