PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS.
Project 1 report

Gabriele Benanti (550552), g.benanti@studenti.unipi.it

September 2024

1 Introduction

The following structure is adopted for the report:
Section 2 provides a concise overview of the Wavefront Computation Problem. Section 3 outlines the sequential approach, with particular attention to the identified bottlenecks. Section 4 and 5 present the Fast Flow and MPI implementations, with a digression on the tests used to compute speedup, scalability and efficiency analysis. Section 6 introduces the Makefile used to compile all the different implementations and the instructions on how to run the code.

2 Wavefront computation description

In the wavefront computation problem, given a square matrix $N \times N$, each element $e^k_{m,m+k}$ of the k-th upper diagonal, with k ranges from 1 to $N-1$ and m ranges from 0 to $N-k-1$, is computed as follows: two vectors are extracted, the first one v^k_m is composed by the elements posed on the same row m with column index ranging from m to $m+k$, while the second vector v^k_{m+k} is formed by the elements posed on the same column $m+k$ with row index ranging from $m+k$ to m. Then scalar product is computed between these two vectors and cubic root is applied to the result. For a visual representation of the computation, refers to Image 1. Since values of elements in the k-th upper diagonal are computed with respect to the values of the elements belonging to previous upper diagonal, in this case ranging from k equals to 0 (main diagonal) to k equals to $k-1$, we can not compute in parallel values of elements belonging to different upper diagonals. Knowing that elements of the main diagonal $e^0_{m,m}$ are initialized with values $(m+1)/n$, with $n = N$, computation are performed starting from the first upper diagonal ($k = 1$) to the last one ($k = N$), that is the upper right corner.
The structure of the problem highlights the fact that, in order to speed up wavefront computation, we need to focus our attention on parallelization strategies involving elements on the same diagonal, rather than elements belonging to different upper diagonals, due to the fact that the computation of elements belonging to a given upper diagonal depends on previous upper diagonal elements. The following sections show how the problem has been tackled, starting with a sequential implementation and moving on to Fast Flow and MPI solutions.

3 Sequential implementation

In the sequential implementation of the wavefront problem, proposed in *ProgWavefront.cpp* file, the program takes in input the dimension of the square matrix, set by default to 512. After initializing the elements of the main diagonal as described in section 2, the computation is encapsulated in the

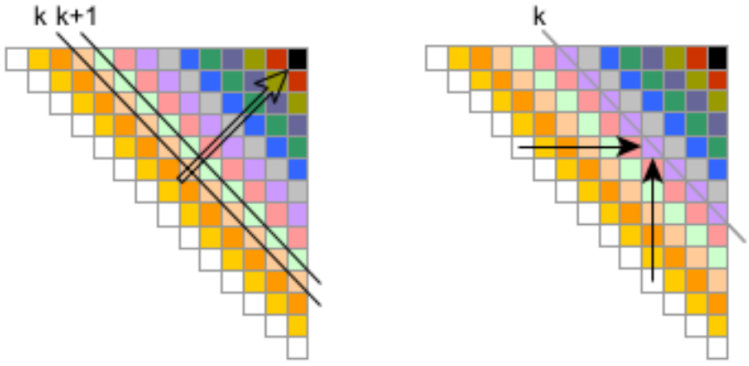

Figure 1: Wavefront computation. Left image: visual representation of wavefront algorithm sequentially. Right image: arrows correspond to the vectors on which scalar product is computed.

wavefront function. This function includes two loop, one inside the other. The outer one scans all the upper diagonal, the inner one scans all the elements of the given k-th upper diagonal, with $k \in \{1, \ldots, n-1\}$. Inside the inner loop, the scalar product is computed incrementally, meaning that we multiply sequentially accessed elements of v_m^k and v_{m+k}^k together and then we sum the result of each multiplication directly in a variable named *dot_product*. In this way we have kept the complexity of the scalar product to $O(k)$. Finally, the result of the scalar product is written in the position corresponding to $e_{m,m+k}^k$. Since the scalar product consists of k multiplications and *k-1* additions, resulting in about 2k Floating Point Operations (FLOPs), and considering that we have to compute *n-k* scalar products for each k-th diagonal, the complexity of the wavefront computation is of the order of $O(n^3)$. To compute the time elapsed during the *wavefront* computation, TIMESTART and TIMESTOP macros, defined in *hpc_helpers.cpp* file, are used.

3.1 Sequential test

To provide more robust results, all the measured times (in seconds) were averaged over 5 runs, enabling -O3 flag for code optimization. All sequential tests were performed on an internal node of the spmcluster machine and on the front-end machine as well. Sequential time for different matrix dimensions are reported in Table 1.
Tests can be run using the SequentialTestFrontend.sh script, which tests front-end performance, or the SequentialTestInternal.sh script, which tests internal nodes. Results of this script are saved in sequential_stat_frontend and sequential_stat_internal respectively.

	1024	2048	4096	8192
$T_{seqInternal}$	0.94s	15.15s	89.35s	630.25s
$T_{seqFrontend}$	0.31s	7.02s	160.75s	1624.42s

Table 1: Average sequential time over 5 runs for different matrix dimensions.

4 Fast Flow implementations

The first attempt at parallelization was carried out using the Fast Flow framework. Before coding a possible parallel implementation, the most important step was to choose the most appropriate parallel pattern to try to reduce the cost of computing elements belonging to the same upper diagonal.
To make this choice the following algorithmic aspects were considered: first, given the structure of the problem, each upper diagonal must be processed sequentially, meaning that computation of elements belonging to k-th upper diagonal must be completed before moving to $k + 1$-th upper diagonal, as represented in Figure 1. Second, since the computation of each element consists only on the execution of a single function, i.e. the scalar product, the workload of a single upper diagonal can be divided among all the available workers.
These observations led to prefer data parallel patterns over stream parallel ones and in particular the **Map** data parallel pattern was chosen accordingly. Since the parallelizable part of the wavefront computation consists of two functions, the first (f) computes an entire upper diagonal, meaning that it computes each element belonging to the given upper diagonal, and the latter (g), computes a scalar product performed on two vectors extracted from the matrix, an expression like $map(f) \circ map(g)$ would have been obtained. Considering that the second function (g) is included in the first one (f), the expression was then refined considering the "map fusion" refactoring rule, obtaining $map(f) \circ map(g) \equiv map(f \circ g)$.
Another important part of the parallel solution design is the scheduling strategy that is used to divide the diagonal elements among the workers and that can lead to a potential load balancing problem and thus further overhead. Since computing an element in the k-th upper diagonal involves only a scalar product of two nonzero vectors of equal length, a static scheduling strategy was used. Therefore, for each upper diagonal, the workload was divided almost equally between the available workers, using a formulation of this kind: *# elements of k-th diagonal / # workers*. If the result of the division is a non-integer number, the modulo operator (%) was used to calculate the remainder of the division, assigning another element to compute to the first T workers, where T is equal to the result of the modulo operation. Static strategies provide a fast way of splitting the workload between the threads, without adding further overheads, for example, due to the management of shared data structures which are typical of dynamic approaches.
Furthermore, a second implementation using Fast Flow **Parallel For** function is proposed. The following sections 4.1 and 4.2 provide a detailed description of the two implementations.

4.1 Map description

The first Fast Flow implementation consists of a Map patter solution, build on top of a master-worker architecture. In particular, the emitter node, a custom instance of type **ff_monode_t**, emits tasks for the workers. Worker nodes, a custom instance of type **ff_node_t**, process these tasks by computing elements along the diagonals of the matrix using a dot product and cube root operation. The calculation of the whole matrix is completed when each worker terminate its execution, and in particular each worker deletes its own task and sends out a **GO_ON** message. The Task data structure includes all the necessary data for computation, such as a **pointer** to the matrix being processed, **start** and **end** variables indicating the interval each worker is responsible for, a variable saving the **number of worker** used, and an **index** variable identifying each worker. Within the *wavefront* function, tasks are created and initialised with the values regarding the first upper diagonal. After the emitter has returned a number of tasks equal to the number of workers, it will send an **EOS** to all workers. In order to keep the workload balanced between the workers throughout the entire wavefront calculation, intervals are recomputed for each worker at the beginning of the following upper diagonal calculation, as described in section 4. Since the worker function **svc** contains the loop that moves over the upper diagonals, we need a synchronisation mechanism to maintain consistent values in the matrix. Therefore, a C++20 barrier was set after processing each diagonal, i.e. at the end of the first for cycle. All the workers

will be exploited throughout the computation, until the condition in which the number of workers is equal to the dimensions of the upper diagonal is reached. Workers with an index greater than the dimension of the current upper diagonal computed will jump directly to the barrier. When the number of workers exceeds the number of physical cores, a function called *no_mapping* is used to disable the explicit assignment of worker threads to specific CPU cores. This approach improves load balancing and resource utilization by giving the operating system more control over thread scheduling.

4.1.1 Map tests

Strong Scaling To assess how the implementation scales with the number of threads, different matrix sizes were tested within the set {1024, 2048, 4096, 8192}. The program was executed both on the front-end node of the cluster and on an internal node using the SLURM *sbatch* command. For each test computed on a different machine, the bash script mapping_string.sh was run on each machine in order to count the logical cores and real cores. Since the number of physical cores is 16 and 20 for the internal node machine and the front-end machine respectively, the number of workers (nw) tested is in the range {1, 2, 4, 8, 16, 20}. Speedup, scalability and efficiency were measured and results for matrices of dimension 2048, 4096 and 8192 are shown in Figure 2, Figure 3 and Figure 4. These plots highlight how performance vary with respect to the matrix dimensions and the machine that execute the tests. When moving from the smaller (2048) to the larger (8192) matrix, performance changes and varies according to the machine running it. In fact, sub-linear speedup is shown in Figure 2 for a matrix of dimension 2048x2048, while it starts to slowly decrease moving to larger matrix dimensions in tests performed on internal machine, as shown in Figure 3 and Figure 4. Performance degradation in internal nodes tests could be due to the fact that, since the scalar product itself is not parallelized, larger matrices have to perform the scalar product on longer vectors, which results in more computation time. Map implementation shows to fully exploit all the physical cores available in both the machines running the tests only on the 8192x8192 matrix dimension. Internal node tests for all the different matrix dimensions also show that all available physical cores are fully utilised. In general, there is a greater performance leap between internal and front-end tests for matrices with dimension larger than 2048x2048.
Tests can be run using the ./FFMapTestFrontend.sh script, which tests front-end performance, or the sbatch FFMapTestInternal.sh script, which tests internal nodes. Other matrix dimensions and number of workers were tested but not reported in the report for reasons of brevity. Results of this script are saved in strong_map_stats_frontend folder and strong_map_stats_internal folder respectively, and are divided per tested configuration. All tests results can be found in Appendix A.1.

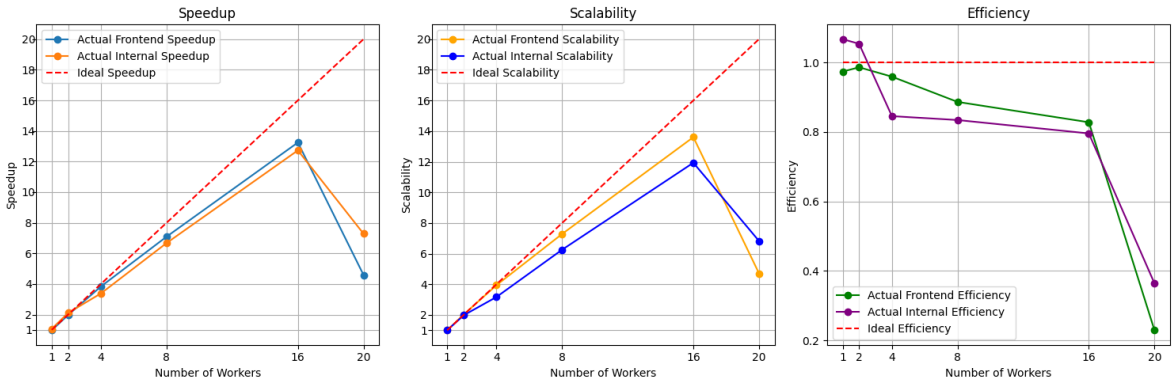

Figure 2: Speedup, scalability and efficiency plots for a 2048x2048 matrix using the Map implementation. Results averaged over 5 runs.

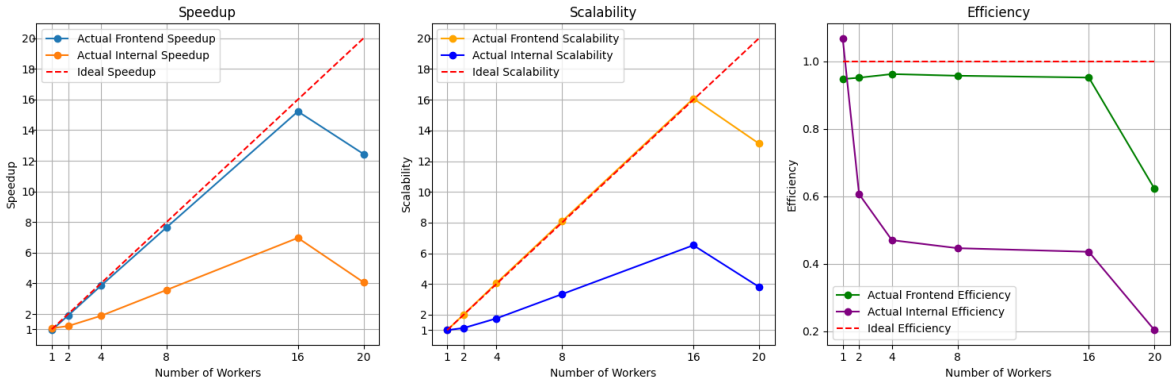

Figure 3: Speedup, scalability and efficiency plots for a 4096x4096 matrix using the Map implementation. Results averaged over 5 runs.

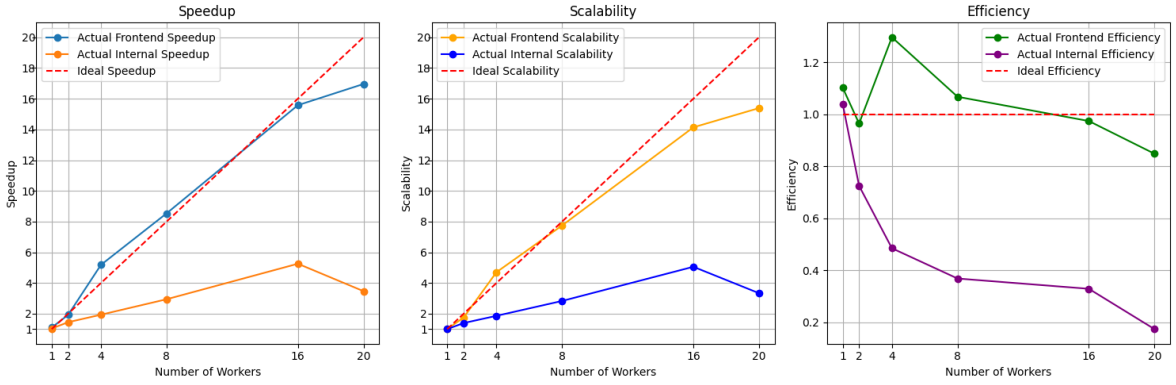

Figure 4: Speedup, scalability and efficiency plots for a 8192x8192 matrix using the Map implementation. Results averaged over 5 runs.

Weak Scaling In the experiment, the total workload is scaled linearly with the number of workers. To do so, following the considerations on the complexity in Section 3, we scale the matrix dimension to $N \times N$, where $N = 2048 \cdot \sqrt[3]{nw}$, and nw is the number of workers. Figure 5 shows how the implementation performs: near linear behaviour is highlighted in the speedup plot for the front-end test, showing differences in performance between the front-end and internal tests. Maximum speedups are achieved with 16 ($N = 5260$) and 20 ($N = 5560$) workers for internal node and front-end machine, respectively. Front-end efficiency increase in the interval between 4 and 16 workers, with N equal to 3250 and 5160 respectively, while internal efficiency remains near almost stable over the same interval. Tests can be replicated using WeakFFMapFrontend.sh and WeakFFMapInternal.sh scripts, and results, divided per tested configurations, are saved in weak_map_stats_frontend and weak_map_stats_internal folders respectively.

4.2 Parallel For description

The second Fast Flow implementation consists of using Parallel For function within *wavefront* function. In particular, the *parallel_for* function was used to parallelise the for cycle that computes each element of a given k-th upper diagonal. Before using it, we initialise an object of class **ParallelForReduce**, with a certain number of workers, at the beginning of the wavefront computation. This object is then

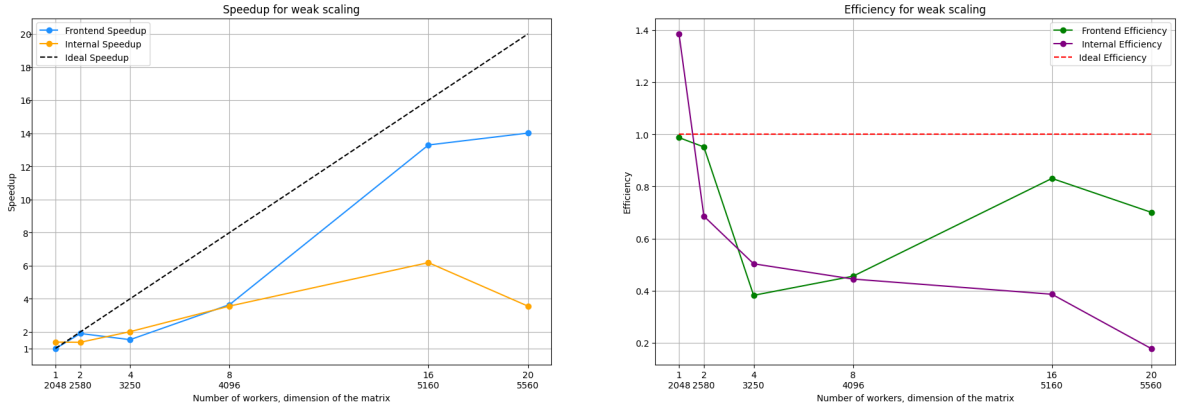

Figure 5: Speedup and Efficiency plots for Weak scaling tests on the Map implementation. Results averaged over 5 runs.

used to parallelise the aforementioned inner for cycle of the *wavefront* function, where each element is calculated as already described in Section 3. Again, static scheduling is used, which means that the workload is divided equally between all available workers. The number of workers is adjusted each time an upper diagonal calculation is started, so that no more workers are used than the number of elements belonging to the current upper diagonal.

4.2.1 Parallel For tests

Strong Scaling The tests performed on the Parallel for implementation are the same as those performed on the Map implementation described in Section 4.1.1. The same considerations already exposed in Section 4.1.1 also apply to this implementation. In this case, physical cores are better exploited in all the tests. Since the *parallel_for* function implementation is based on a map, similar performance is expected in the tests. In fact, Figure 6, Figure 7 and Figure 8 show similar performance with respect to the map implementation. Again, there are significant performance differences between code running on the front-end machine and code running on the internal node.
Tests can be run using the ./FFParForTestFrontend.sh script, which tests front-end performance, or the sbatch FFParForTestInternal.sh script, which tests internal nodes. Other matrix dimensions and number of workers were tested but not reported in the report for reasons of brevity. Results of this script are saved in parfor_stats_frontend folder and parfor_stats_internal folder respectively. All tests results can be found in Appendix A.2.

Weak Scaling Same tests, as the one described for the Map implementation, were performed on the Parallel For implementation. Figure 9 shows similar performance with respect to the Map implementation, with just a small upgrade for the front-end speedup and efficiency.
Tests can be replicated using WeakFFParForFrontend.sh and WeakFFParForInternal.sh scripts, and results, divided per configurations tested, are saved in weak_parfor_stats_frontend and weak_parfor_stats_internal folders respectively.

5 MPI implementation

The MPI implementation, unlike the Fast Flow implementation, allows the workload to be distributed across multiple MPI processes running on a cluster of machines. For this implementation, instead of using a typical master-slayer architecture, where a master process sends pieces of the matrix to be computed to all slayer processes, each worker, i.e. an MPI process, maintains a local copy of

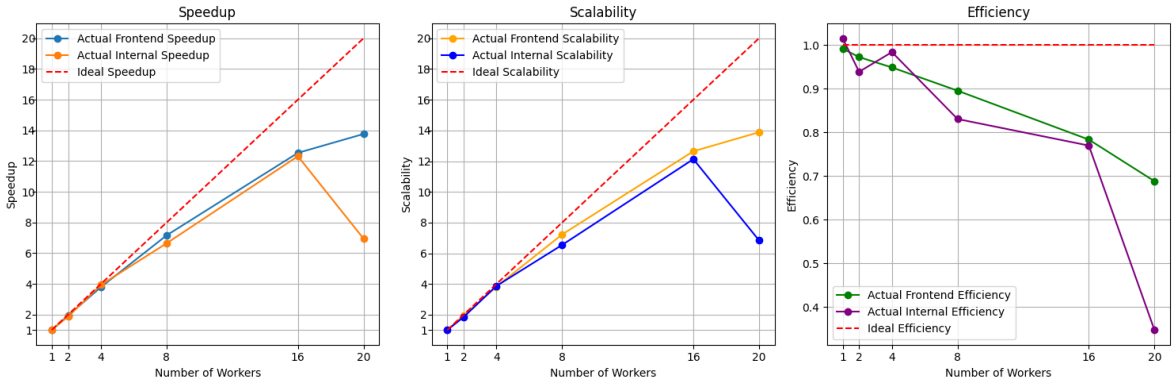

Figure 6: Speedup, scalability and efficiency plots for a 2048x2048 matrix using the Parallel For implementation. Results averaged over 5 runs.

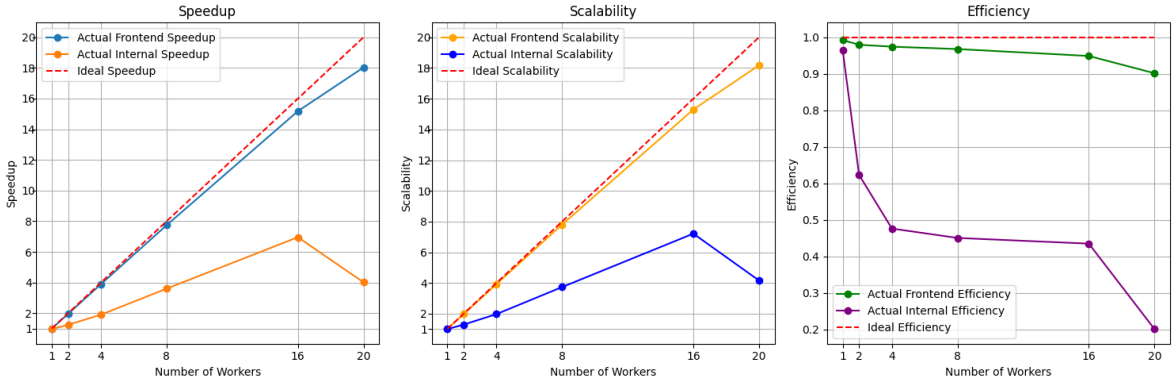

Figure 7: Speedup, scalability and efficiency plots for a 4096x4096 matrix using the Parallel For implementation. Results averaged over 5 runs.

the matrix to be computed and operates on a different part of each upper diagonal. The workload was divided between workers in a similar way to that used in the Fast Flow Map implementation. Each MPI process operates on a different interval of the k-th diagonal, and each interval was defined using the same idea proposed in Section 4, meaning that the workload was kept almost equal between workers until the length of the k-th diagonal was greater than or equal to the number of workers. In order to maintain consistent results throughout the wavefront computation, all the processes must communicate each other. For this reason, the *MPI_Allgatherv* function was used at the end of each diagonal computation. This function performs a collective communication operation where data from all processes in a communicator is gathered and distributed to all processes. Each process can send data of different sizes, and *MPI_Allgatherv* accommodates this by allowing to specify the number of elements contributed by each process, as well as where the received data should be stored in each process's receive buffer. The function only returns once all the data has been gathered and distributed among all participating processes, meaning that a synchronization mechanism is implemented in the function itself, so there is no reason to use a barrier. For each upper diagonal calculation, two integer vectors were used to save offsets and displacements for all participating processes, and two double vectors were used to hold the result computed in each process and to collect all results from all processes.

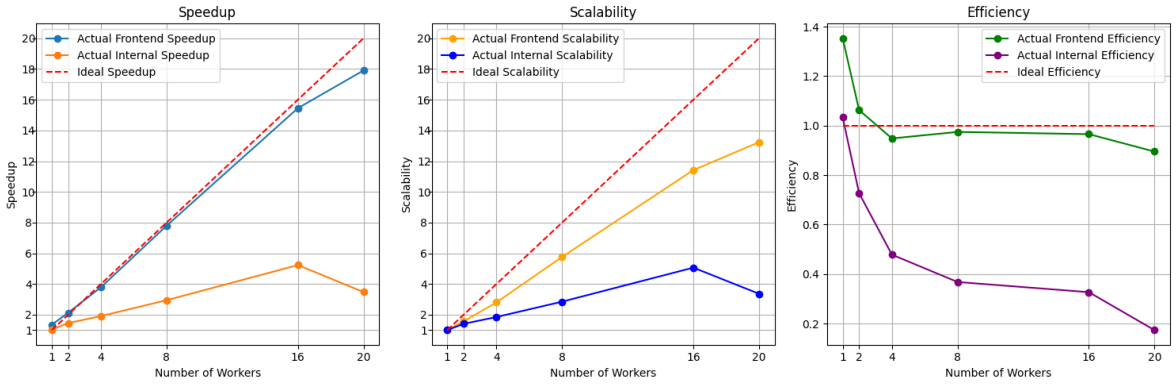

Figure 8: Speedup, scalability and efficiency plots for a 8192x8192 matrix using the Parallel For implementation. Results averaged over 5 runs.

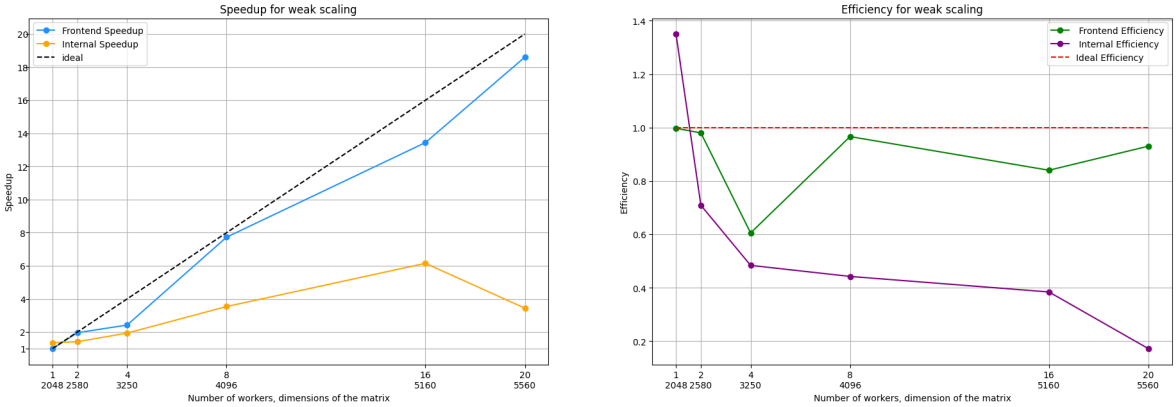

Figure 9: Speedup and Efficiency plots for Weak scaling tests on Parallel For implementation. Results averaged over 5 runs.

5.1 MPI tests

The MPI implementation was tested on the internal nodes of the **spmcluster** machine, testing different configurations with different numbers of processes (p), nodes (n) and matrix dimensions (N). In each test, the matrix dimension was varied, with $N \in \{1024, 2048, 4096, 8192\}$, and different numbers of processes, with $p \in \{1, 2, 4, 8, 16, 32, 64, 128\}$, were used on the internal nodes available, with $n \in \{1, 2, 4, 6, 8\}$. Figure 10, Figure 11 and Figure 12 show speedup, scalability and efficiency plots for different matrix dimensions. Tests results show higher speedup for smaller matrices, until $p \leq 16$, and a higher speedup for larger matrices, starting from $p \geq 32$; scalability and efficiency performance also show the same trend. This behaviour could be due to the fact that large matrices make better use of a larger number of processes, while small matrices exploit better fewer processes. The general performance of the metrics deviates more from the ideal one starting from $p \geq 32$ for $N = 2048$ and $p \geq 16$ for $N = \{4096, 8192\}$. This may be due to the large overhead of creating and managing so many processes on different machines. Another limitation of this implementation is that each process must maintain a copy of the full matrix in memory, which can lead to machine memory saturation for large matrix dimensions. Nevertheless, the MPI implementation, tested with 128 processes on 8 nodes, allows to reduce the execution time, for a matrix of $N = 8192 \times 8192$, from 630s to 25s.
Tests can be replicated using `sbatch MPItest.sh` script, which contains also $N = 1024$ as a matrix

dimension. Results of each configuration tested are saved in MPItests folder. Mean and variance values can be extracted using calculate_stats.sh script, and results for each configuration are saved in MPIstats folder. An example of the script, containing all the SLURM specification used to run the MPI implementation, is present in mybash.sh file. All tests results can be found in Appendix B.

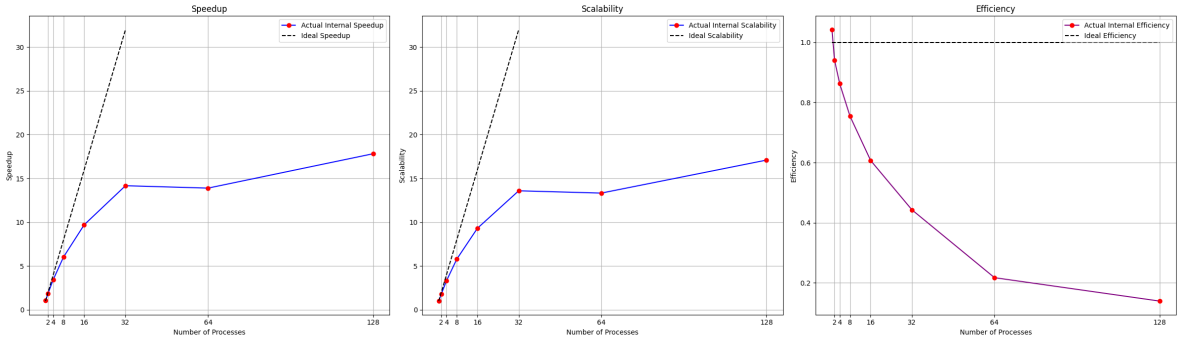

Figure 10: Speedup, scalability and efficiency plots for a 2048x2048 matrix using the MPI implementation. Results averaged over 5 runs.

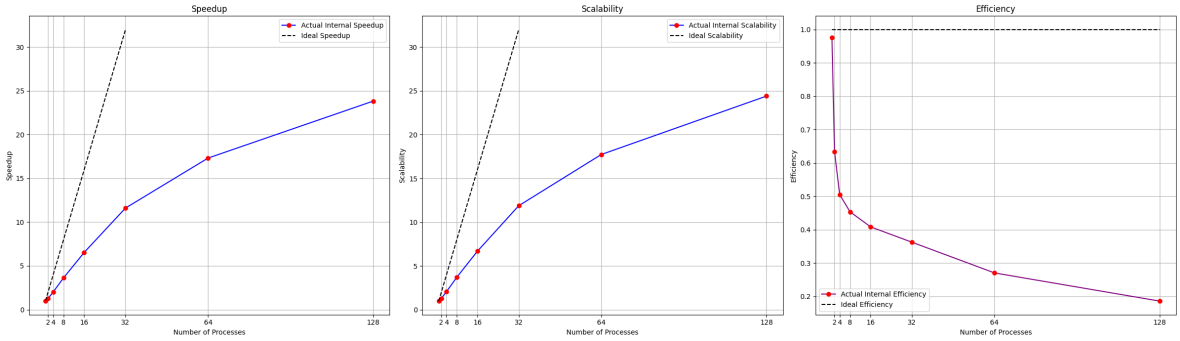

Figure 11: Speedup, scalability and efficiency plots for a 4096x4096 matrix using the MPI implementation. Results averaged over 5 runs.

6 Compilation and execution

6.1 Sequential implementation

To compile the sequential code, there is a Makefile in the Sequential folder. The program can be executed with ./ProgWavefront N for the front-end machine or srun -N 1 ProgWavefront N for the internal machine, where N is the dimension of the matrix.

6.2 Fast Flow implementation

To compile the two implementations, a Makefile is present in the FastFlow folder, which contains -03 and -ffast-math flags. Map implementation can be executed with ./MapFF N p for the front-end machine or srun -N 1 MapFF N p for the internal machine, where N is the dimension of the matrix and p is the number of workers.

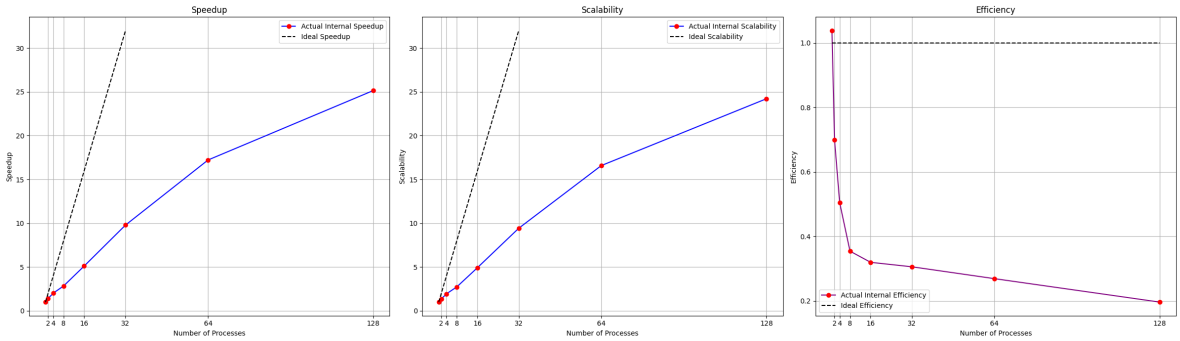

Figure 12: Speedup, scalability and efficiency plots for a 8192x8192 matrix using the MPI implementation. Results averaged over 5 runs.

Parallel For implementation can be executed with ./ParForFF N p for the front-end machine or srun -N 1 ParForFF N p for the internal machine.

6.3 MPI implementation

For the MPI implementation, a Makefile is posed in the MPI folder, which contains -03 flag. The code can be executed in the internal machines with sbatch mybash.sh, which contains srun command to execute the code with all the parameters needed, such as matrix dimension, number of processes and number of nodes.

A Fast Flow execution times

A.1 Map implementation times

Front-end machine				
N. workers	**Matrix dimension**			
	1024	**2048**	**4096**	**8192**
1	0.292s	7.215s	169.786s	1474.112s
2	0.148s	3.566s	84.518s	842.584s
4	0.075s	1.831s	41.790s	313.425s
8	0.040s	0.993s	21.001s	190.328s
16	0.029s	0.532s	10.561s	104.285s
20	0.103s	1.535s	12.919s	95.793s
32	0.081s	1.066s	14.189s	129.517s
40	0.084s	1.006s	15.562s	106.743s

Table 2: Execution time for different matrix sizes and number of workers for Map implementation running on the front-end machine. Results averaged over 5 runs.

Internal node machine				
N. workers	**Matrix dimension**			
	1024	**2048**	**4096**	**8192**
1	0.951 s	14.207s	65.699s	606.496s
2	0.512s	7.190s	75.484s	434.961s
4	0.268s	4.489s	48.679s	325.266s
8	0.146s	2.271s	25.638s	214.077s
16	0.086s	1.193s	13.135s	119.919s
20	0.209s	2.087s	22.481s	181.664s
32	0.155s	1.371s	15.095s	130.772s
48	0.219s	1.655s	18.480s	156.376s
64	0.219s	1.665s	17.470s	145.276s

Table 3: Execution time for different matrix sizes and number of workers for Map implementation running on the internal node machine. Results averaged over 5 runs.

A.2 Parallel For implementation times

Frontend machine				
N. workers	**Matrix dimension**			
	1024	**2048**	**4096**	**8192**
1	0.291s	7.086s	162.574s	1200.724s
2	0.157s	3.611s	81.826s	764.197s
4	0.085s	1.852s	41.334s	428.298s
8	0.063s	0.564s	20.946s	208.342s
16	0.059s	0.517s	10.557s	105.149s
20	0677s	0.558s	8.916s	90.728s
32	0.107s	1.072s	14.215s	120.384s
40	0.124s	1.075s	11.542s	185.636s

Table 4: Execution time for different matrix sizes and number of workers for Parallel For implementation running on the frontend machine. Results averaged over 5 runs.

Internal node machine				
N. workers	**Matrix dimension**			
	1024	**2048**	**4096**	**8192**
1	0.940s	14.929s	94.885s	599.264s
2	0.550s	8.076s	73.441s	433.325s
4	0.297s	3.858s	48.043s	329.143s
8	0.169s	2.288s	25.392s	213.936s
16	0.112s	1.234s	13.151s	120.389s
20	0.190s	2.181s	22.780s	181.213s
32	0.166s	1.326s	15.418s	130.612s
48	0.179s	1.350s	14.236s	126.033s
64	0.226s	1.471s	15.175s	129.073s

Table 5: Execution time for different matrix sizes and number of workers for Parallel For implementation running on the internal machine. Results averaged over 5 runs.

B MPI execution times

Nuomber of Nodes = 1				
N. processes	**Matrix dimension**			
	1024	**2048**	**4096**	**8192**
1	**0.955s**	**14.530s**	**91.510s**	**606.883s**
2	**0.493s**	8.127s	92.846s	672.024s
4	**0.286s**	4.459s	50.338s	468.788s
8	**0.195s**	**2.374s**	25.795s	238.811s
16	**0.094s**	**1.228s**	13.157s	122.976s
32	**0.177s**	1.631s	16.763s	168.052s

Table 6: Execution time for different matrix sizes and number of processes on 1 node. Results in bold are used to calculate the metrics. Results averaged 5 times.

Number of Nodes = 2				
N. processes	**Matrix Dimensions**			
	1024	**2048**	**4096**	**8192**
2	0.577s	**8.050s**	70.43s	450.207s
4	0.355s	**4.334s**	47.5434s	430.552s
8	0.315s	2.503s	25.286s	240.685s
16	0.235s	1.519s	13.694s	122.754s
32	0.275s	**1.069s**	7.793s	66.964s

Table 7: Execution time for different matrix sizes and number of processes on 2 node. Results in bold are used to calculate the metrics. Results averaged 5 times.

Number of Nodes = 4				
N. processes	**Matrix Dimension**			
	1024	**2048**	**4096**	**8192**
2	0.576s	8.291s	**68.13s**	**445.795s**
4	0.385s	4.393s	**44.192s**	**312.169s**
8	0.337s	2.518s	**24.613s**	**222.296s**
16	0.264s	1.567s	**13.677s**	**123.178s**
32	0.282s	1.156s	7.71631s	65.492s
64	0.402s	**1.094s**	5.090s	38.701s

Table 8: Execution time for different matrix sizes and number of processes on 4 node. Results in bold are used to calculate the metrics. Results averaged 5 times.

Number of Nodes = 6				
N. processes	**Matrix Dimension**			
	1024	**2048**	**4096**	**8192**
32	0.331s	1.192s	7.773s	64.519s
64	0.413s	1.197s	5.201s	37.587s
128	0.357s	1.239s	6.874s	51.147s

Table 9: Execution time for different matrix sizes and number of processes on 6 node. Results in bold are used to calculate the metrics. Results averaged 5 times.

Number of Nodes = 8				
N. processes	**Matrix Dimension**			
	1024	**2048**	**4096**	**8192**
32	0.317s	1.167s	**7.706s**	**64.377s**
64	**0.376s**	1.160s	**5.029s**	**36.313s**
128	**0.326s**	**0.857s**	**3.738s**	**24.832s**
256	0.488s	1.358s	5.386	40.143s

Table 10: Execution time for different matrix sizes and number of processes on 8 node. Results in bold are used to calculate the metrics. Results averaged 5 times.