SPM Report

Leonardo Crociani - 615392

January 16, 2025

Contents

1 Introduction

This report presents the work on the first project: wavefront computation. The following assumptions were made:

- Both `matrix_size` and `num_workers`/`num_processes` are considered to be of type `size_t`.
- The `(i,j)` elements of the `k`-th diagonal are computed as illustrated in the diagram below:

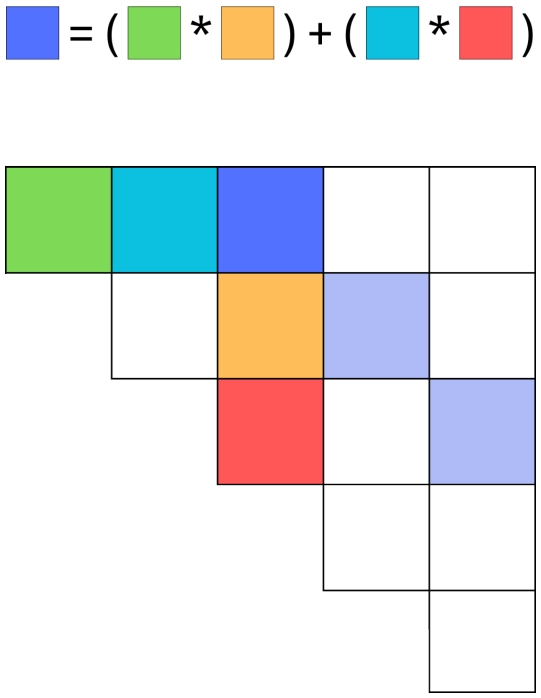

Figure 1: Wavefront single element computation

2 Code Structure

The delivered zip file contains the following files and directories:

- `src`: This directory includes the source files:
  - `./run-all.sh`: A bash script required to install the *FastFlow* library and build the code. It can also be used to run each version as a test.
  - `./run.sh`: Another script usable to run a single version of the code with different parameters. Check section 3.0.2 for more details.
  - `sequential.cpp`, `fastflow.cpp`, and `mpi.cpp`: These files implement the functionalities required by the project specifications.
  - `Makefile`: Used for building the code.
  - `lib`: A folder that contains the *FastFlow* library (after installation) and all other utility `.hpp` files.
  - `jobs`: A folder containing the `sbatch` job scripts used to execute the three versions when launching the `run-all.sh` script. [1]
- `scripts`: This folder contains Python scripts used to measure the performance of each code version by varying parameters (`#PEs`[2], `matrix_size`) and to plot the data. Check out section 3.0.3 to learn more.

[1] The `sbatch` command was utilized to facilitate parameter reuse rather than for its deferred execution feature.
[2] Processing Entities

3 Running the Code

3.0.1 Testing Once

To test the code, follow these steps:

1. Open a shell and navigate to the unzipped folder[3].
2. Run `cd src && ./run-all.sh`. The script will:
   (a) Clone the *FastFlow* repository into the `lib` folder.
   (b) Run `mapping_string.sh`.
   (c) Build the code.
   (d) Prompt the user to execute the code once for testing purposes[4].

3.0.2 Testing with Different Parameters Manually

From the `src` directory you can manually test the code with different parameters by running the script `run.sh`. It builds the code and uses the following parameters:

- `./run.sh <target> <parameters>`
- `<target>` can be `sequential`, `fastflow` or `mpi`
- `<parameters>` depends on the target:
  - Sequential usage: `./run.sh sequential <matrix_size>`
  - FastFlow usage: `./run.sh fastflow <matrix_size> <num_workers>`
  - MPI usage: `./run.sh mpi <matrix_size> <num_processes_per_node> <num_nodes>`

Each program will print the execution time (in milliseconds) and the value of the top-right element of the matrix.

3.0.3 Running Extensive Measurements and Plotting Results

To run extensive tests with predefined parameter ranges, record the results, and plot them, follow these steps [5]:

1. Open a shell in the root folder.
2. Navigate to the `scripts` directory.
3. Run `./start.sh`. The script will:
   (a) Create a Python virtual environment.
   (b) Download and install the required dependencies.
   (c) Execute the code (using `srun`) and measure performance.
   (d) Generate plots of the results.
4. The results will be saved in `scripts/data/results`.
5. The plots will be saved in `scripts/data/plots`.

[3]Alternatively, clone the repository directly from GitHub using the command:
`git clone https://github.com/leonardocrociani/Parallel-Distributed-Stencil-Computation.git && cd Parallel-Distributed-Stencil-Computation`

[4]The test uses `matrix_size = 1024`, `#processes` = 4 (2 per node) and `#workers` = 4.

[5]This will take hours.

4 Parallelization Strategy

4.1 FastFlow

For the FastFlow version, I chose to use an `ff_Farm` since:

- we can see the progressive diagonal computation as a stream
- we have to compute the same algorithm for each element of this stream

The farm consists of an emitter and `<num_workers>` workers (specified via a CLI parameter). Each worker's output is reconnected to the emitter's input for synchronization purposes.

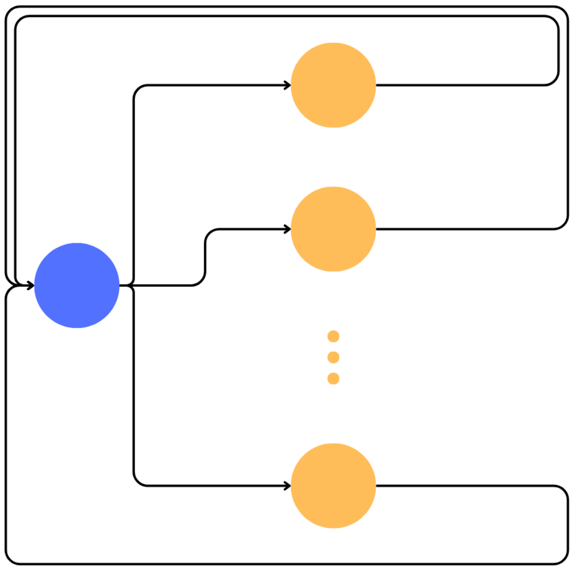

Figure 2: Farm schema. The blue node represents the emitter, while the other nodes are the workers.

4.1.1 Execution and Synchronization

The computation of the diagonals must be synchronized, as each step requires the completion of the previous one. To achieve this:

- The emitter tracks the number of active workers.
- The emitter assigns tasks, defined as a range of rows to process. Since these ranges do not overlap, access to the matrix is lock-free, implemented using a pointer.
- Once a worker computes the elements in its assigned range, it returns the task via the channel to the emitter.
- The emitter then decreases the count of active workers.
- If no active workers remain and the iterations are not complete, it implies that all workers have finished their tasks for the current iteration, allowing the next iteration to begin.

4.1.2 Workload Balancing

Since the workload per iteration is fixed, I implemented a block task distribution.

4.2 MPI

The MPI version leverages collectives to reduce the communication overhead introduced by multiple `MPI_Send` and `MPI_Recv` calls. In this implementation, there is no master process; every process acts as an active worker. In the tests I assigned two processes [6] to each cluster node .

4.2.1 Execution and Synchronization

After initializing the matrix, each process performs the following steps until the wavefront computation is complete:

1. Calculate the range of action for each process based on its rank.
2. Compute local partial results.
3. Compute the displacements and receive counts required for inter-process communication.
4. Use `MPI_Allgatherv` to exchange the newly computed elements among all processes.
5. Update the local matrix with the received data.

At the end of the computation, an `MPI_Reduction` using the `MPI_MAX` operation ensures the maximum computation time among all processes is recorded. This value is printed by the process with rank 0.

4.2.2 Workload Balancing

The range of action for each process is computed using a static block distribution, similar to the FastFlow implementation.

5 Performance Analysis and Plots

The base matrix side size is set to 512.

5.1 Sequential Version

For the sequential version, I tested the implementation using matrix side sizes defined as $512 * i$ for $i \in \{2, 4, \ldots, 20\}$.

The execution time (measured in milliseconds[7]) follows an exponential trend, as shown in the plot below and reported in the table 1:

Matrix Dimension	Time (ms)
1024	1042.97126
2048	16169.97706
3072	47328.08337
4096	101843.43242
5120	160846.40532
6144	276178.04761
7168	436739.30767
8192	610843.9207
9216	881424.21685
10240	1199850.19243

Table 1: Sequential Times with different Matrix Dimensions

[6] Each SPMcluster node has 20 physical cores. However, under weak scaling conditions, the matrix side size is calculated as `#processes` $* 512$, where 512 is the base matrix side size. For 8 processes and 20 cores per node, this results in a matrix of size 71680×71680, which is computationally prohibitive; this is the cause I considered 2 processes for each node.

[7] Timing measurements are implemented in `src/lib/chronometer.hpp`.

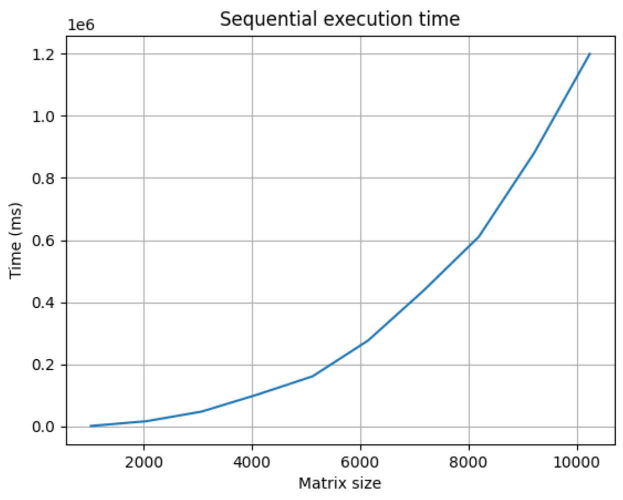

Figure 3: Execution times for the sequential version.

5.2 FastFlow Version

For the FastFlow implementation, I evaluated combinations of the following parameters:

- Number of workers: This parameter varies from 1 to 19. Note that at each iteration the total number of threads is `num_workers` + 1, as one thread is dedicated to the emitter. The upper limit of workers was selected to ensure one thread per core.
- Matrix side size: Defined as $512 * i$, where $i \in \{2, 4, \ldots, 20\}$.

Table 5.2 shows the best execution times (rounded to whole numbers) for each combination of the number of threads (**T**) and matrix side size (**M**):

T\M	**1024**	**2048**	**3072**	**4096**	**5120**	**6144**	**7168**	**8192**	**9216**	**10240**
2	1114	14902	48910	99363	164817	297235	440848	646837	921402	1266711
4	436	5971	23142	62208	109629	179327	266103	379484	492819	667014
6	272	3588	12847	38495	73258	128927	200959	297447	398232	529002
8	202	2580	10203	29412	57165	102198	163023	243796	340933	467263
10	383	4408	19298	47171	81188	132952	196820	322756	372588	478788
12	315	2673	14374	39809	68764	136622	174114	235586	314241	417877
14	289	1672	12924	29241	58749	97385	154319	210574	294036	394927
16	233	2605	11035	28435	59054	101786	144644	191664	266164	430169
18	203	2254	9764	26454	49005	86277	130839	181899	255459	342577
20	202	2072	8650	24159	49103	84774	137558	199351	282908	443005

Table 2: Execution times (in milliseconds) for various matrix side sizes and threads counts.

5.2.1 Strong Scaling

The speedup and efficiency metrics for strong scaling are plotted for two fixed matrix side sizes: 2048 and 5120.

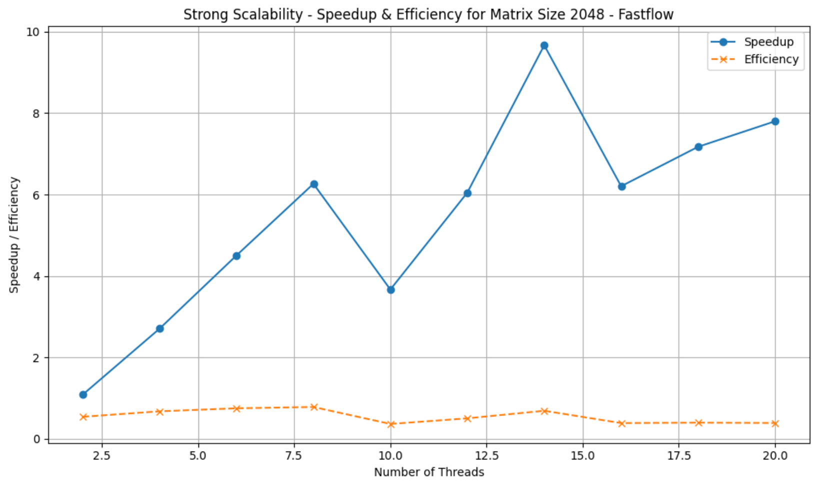

Figure 4: FastFlow-based version speedup and efficiency under strong scaling: 2048 matrix side size.

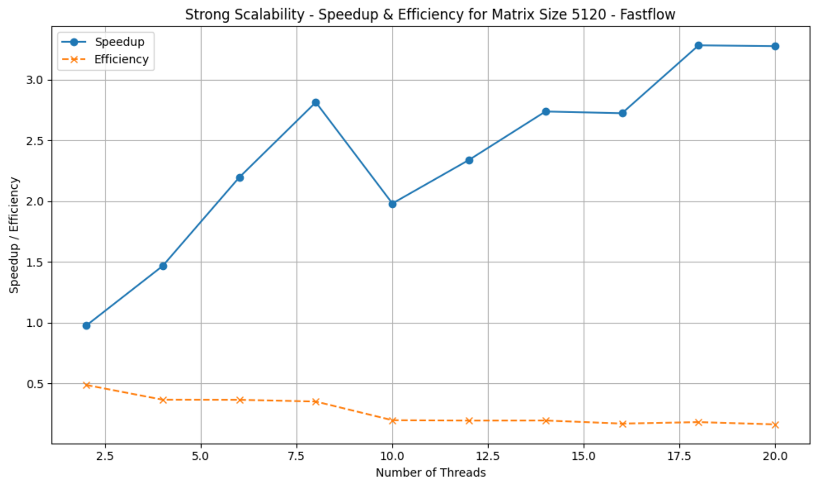

Figure 5: FastFlow-based version speedup and efficiency under strong scaling: 5120 matrix side size.

5.2.2 Weak Scaling

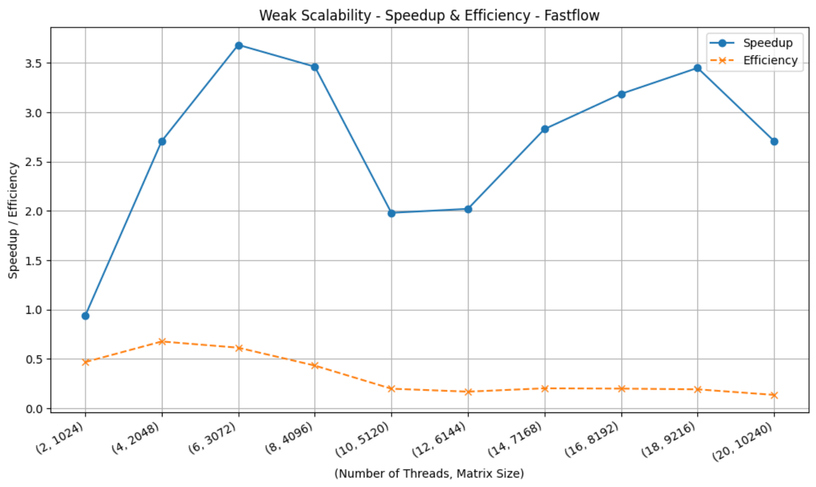

Figure 6: FastFlow-based version speedup and efficiency under weak scaling: variable matrix side size.

Under strong scaling (fig. 4, 5), for smaller matrix sizes (2048), speedup peaks sharply but fluctuates, suggesting inefficiencies at higher thread counts. Larger matrices (5120) scale more smoothly, though efficiency remains low (≈10–20%).

In terms of weak scaling (fig. 6), the FastFlow version shows reasonable speedup, peaking at ≈ 3.5x, but efficiency declines consistently, stabilizing around 20–30%, highlighting scalability limitations due to overhead as thread count grows.

5.3 MPI Version

For the MPI implementation, the following configurations were tested:

- Number of processes: Each node was assigned 2 processes, utilizing all available nodes in the *spmcluster* (8 nodes in total). Thus, the number of processes used was $i \in \{2, 4, \dots, 16\}$.
- Matrix side size: Scaled with the number of processes, following the formula $512 * i$, where i represents the number of processes at each iteration, i.e., $i \in \{2, 4, \dots, 16\}$.

The table below presents the execution times for different combinations of processes (**P**) and matrix side sizes (**M**):

P\M	**1024**	**2048**	**3072**	**4096**	**5120**	**6144**	**7168**	**8192**
2	564	8247	32467	94337	183558	303188	463261	627040
4	377	4419	16608	48126	95827	174395	282945	441355
6	315	3104	11423	33036	64793	117675	191876	298824
8	347	2565	8818	24893	49142	88701	144577	225539
10	313	2202	7402	20491	40077	71830	116756	182192
12	304	1914	6324	17226	33842	60871	98200	152717
14	296	1711	5539	15133	29287	52907	84663	131938
16	276	1599	4958	13359	25814	46219	74406	115918

Table 3: Execution times (in milliseconds) for various matrix side sizes and processes counts.

5.3.1 Strong Scaling

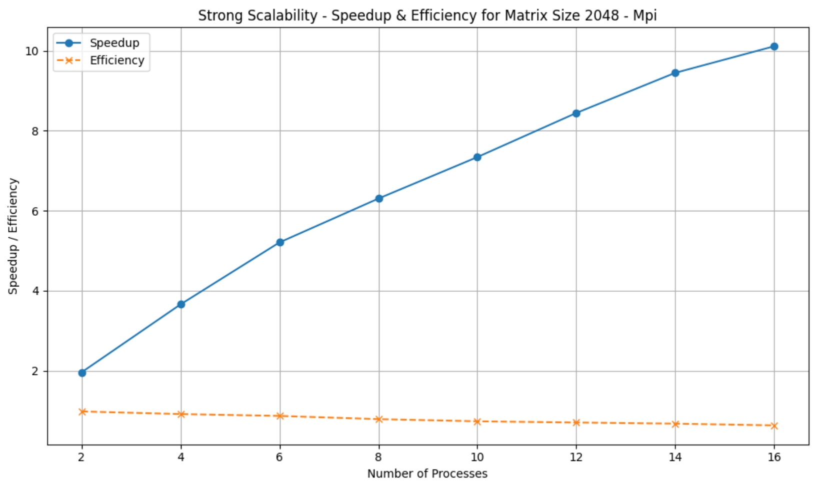

Figure 7: MPI-based version speedup and efficiency under strong scaling: 2048 matrix side size.

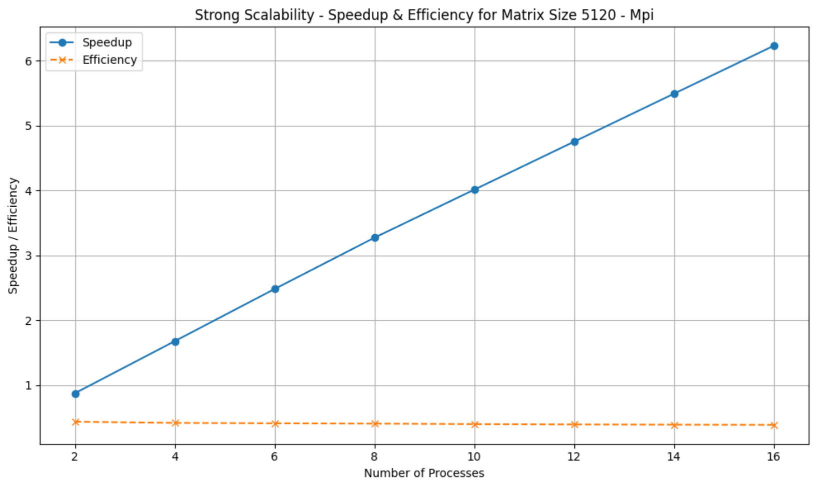

Figure 8: MPI-based version speedup and efficiency under strong scaling: 5120 matrix side size.

5.3.2 Weak Scaling

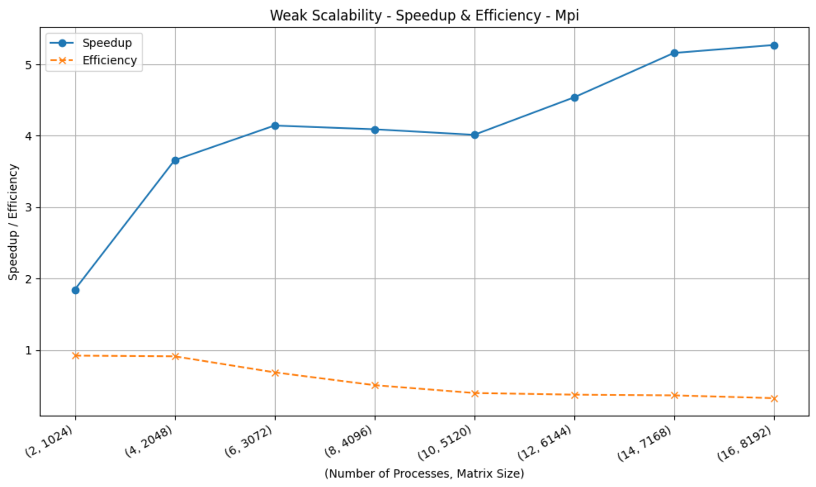

Figure 9: MPI-based version speedup and efficiency under weak scaling: variable matrix side size.

Strong scaling (fig. 7 and 8) shows near-linear speedup but a decline in efficiency as processes increase, with better performance for larger matrices. Weak scaling (fig. 9) demonstrates an increasing speedup that appears to stabilize, accompanied by a gradual decline in efficiency as the problem size scales proportionally with the number of processes.

6 Comparison and Conclusion

The following tables summarize the performance metrics (time in milliseconds) for the top 3 configurations with the best speedup (and different matrix side sizes) for both the FastFlow and MPI versions.

6.1 FastFlow Version

Matrix side Size	Threads	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
2048.0	14	1672.54	$\frac{T(1)}{T(p)} = \frac{16169.98}{1672.54} = 9.67$	$\frac{S(p)}{p} = \frac{9.67}{14} = 0.69$
3072.0	20	8650.84	$\frac{T(1)}{T(p)} = \frac{47328.08}{8650.84} = 5.47$	$\frac{S(p)}{p} = \frac{5.47}{20} = 0.27$
1024.0	20	202.46	$\frac{T(1)}{T(p)} = \frac{1042.97}{202.46} = 5.15$	$\frac{S(p)}{p} = \frac{5.15}{20} = 0.26$

Table 4: Top 3 Configurations with the Best Speedup for FastFlow-based Version

6.2 MPI Version

Matrix side Size	Processes	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
2048.0	16	1599.04	$\frac{T(1)}{T(p)} = \frac{16169.98}{1599.04} = 10.11$	$\frac{S(p)}{p} = \frac{10.11}{16} = 0.63$
3072.0	16	4958.02	$\frac{T(1)}{T(p)} = \frac{47328.08}{4958.02} = 9.55$	$\frac{S(p)}{p} = \frac{9.55}{16} = 0.60$
4096.0	16	13359.07	$\frac{T(1)}{T(p)} = \frac{101843.43}{13359.07} = 7.62$	$\frac{S(p)}{p} = \frac{7.62}{16} = 0.48$

Table 5: Top 3 Configurations with the Best Speedup for MPI-based Version

6.3 Analysis

As seen in the tables and through the plots, the FastFlow and MPI versions exhibit distinct performance characteristics under both strong and weak scaling tests.

- Under **strong scaling**, the FastFlow version shows sharper speedup for smaller matrices but fluctuates at higher thread counts. The MPI version demonstrates near-linear speedup for larger matrices, with efficiency decreasing more gradually as processes increase.
- For **weak scaling**, the FastFlow version achieves reasonable speedup (approximately 3.5x) but experiences a more pronounced efficiency decline. The MPI version shows stabilizing speedup with a slower drop in efficiency, though communication overhead becomes noticeable at scale.