SPM Report

Leonardo Crociani - 615392

January 7, 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:

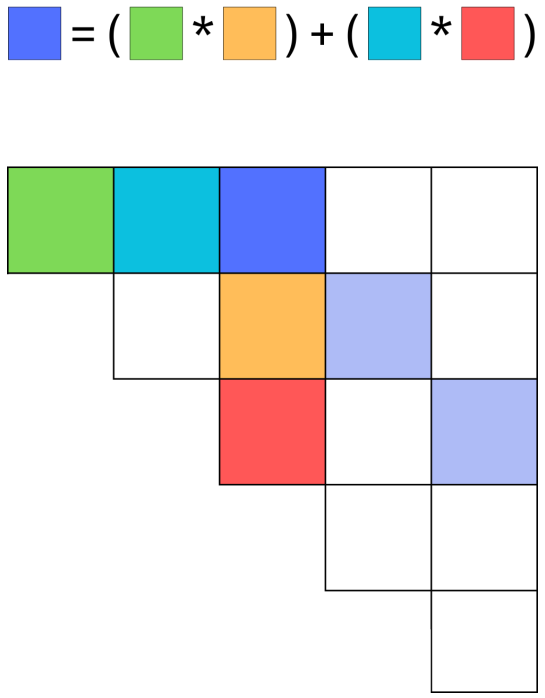

Figure 1: Wavefront single element computation

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. 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 3.0.3 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 that includes the sbatch job to run the MPI version.
- `scripts`: This folder contains Python scripts used to measure the performance of each code version by varying parameters (`#workers`, `#processes`, `matrix_size`) and to plot the data.

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[1].
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[2].

3.0.2 Running Extensive Measurements and Plotting Results

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

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` for the MPI version) 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`.

3.0.3 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_threads> <num_nodes>`

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

[1] 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`

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

4 Parallelization Strategy

4.1 FastFlow

For the FastFlow version, I chose to use an `ff_farm`. It 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.

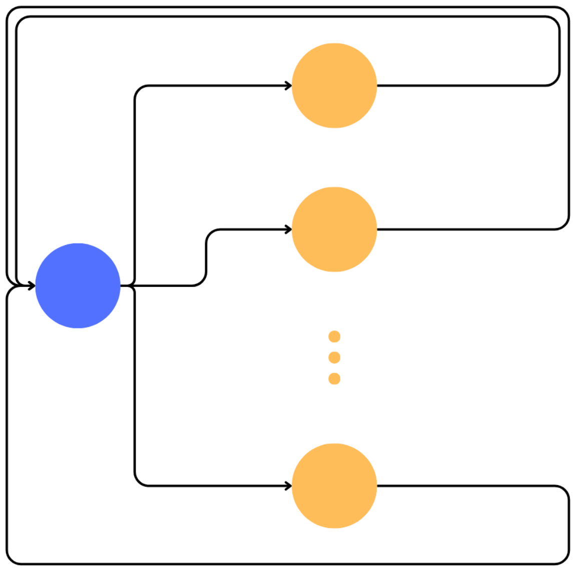

Figure 2: Farm schema. The blue node represents the emitter, while the other nodes are the workers.

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. Additionally, I assigned one process to each cluster node and used `OpenMP` to parallelize the main computation loop within each 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 based on the process rank and compute local partial results using `openmp parallel for`, setting `num_threads(20)`[3].
2. Compute the displacements and receive counts needed for message passing between processes.
3. Perform an `MPI_Allgatherv` to exchange newly computed elements with other processes.
4. Update the local matrix values.

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. Among threads in the `omp parallel` loop, the static scheduling policy is used.

5 Performance Analysis and Plots

The base matrix size is set to 512.

5.1 Sequential Version

For the sequential version, I tested the implementation using matrix sizes defined as $512 * i$ for $i \in \{1, \ldots, 10\}$.

The execution time (measured in milliseconds[4]) follows an exponential trend, as shown in the plot below:

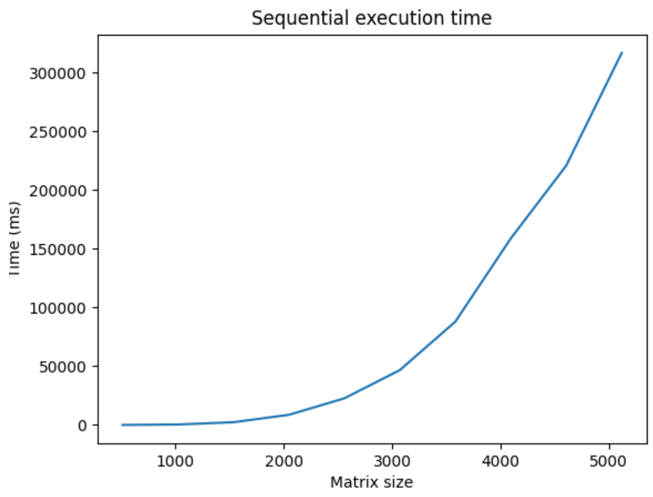

Figure 3: Execution times for the sequential version.

[3]Each SPMcluster node has 20 physical cores and 40 logical cores. Empirically, using hyperthreading slightly worsened performance.Additionally, the number of threads is specified through the `OMP_NUM_THREADS` environment variable.

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

5.2 FastFlow Version

For the FastFlow version, I explored combinations of the following parameters:

- Number of workers: The number of workers ranges from 1 to 10. The upper limit of 10 workers is chosen to maintain a reasonable balance between matrix size and computation time, as weak scaling increases the matrix size proportionally with the number of workers.
- Matrix size: Defined as $512 * i$ for $i \in \text{range}(1, \text{max\_num\_workers} + 1)$.

Table 1 shows the best execution times (rounded to whole numbers) for each combination of the number of workers (**W**) and matrix size (**M**):

W\M	**512**	**1024**	**1536**	**2048**	**2560**	**3072**	**3584**	**4096**	**4608**	**5120**
1	48	409	2390	8594	23275	47933	89733	161736	223975	324772
2	24	208	1207	4914	11630	24030	45096	81927	114866	162196
3	17	141	807	3643	7730	16138	30067	54495	76823	109261
4	13	107	620	2731	5793	12104	22698	40989	57606	81808
5	11	86	497	2228	4604	9851	18309	32815	46290	65806
6	9	73	417	1941	3965	8250	15272	27277	38982	54446
7	9	64	357	1685	3422	7155	13037	23441	32878	46933
8	8	56	315	1496	2992	6144	11432	20674	28946	41563
9	7	50	282	1377	2625	5573	10305	18454	25626	36781
10	7	47	257	1232	2399	4959	9147	16520	23246	33248

Table 1: Execution times (in milliseconds) for various matrix sizes and worker counts. The best time in each column is highlighted.

5.2.1 Strong Scaling

The speedup and efficiency metrics for strong scaling are plotted for two fixed matrix sizes: 1024 and 2048.

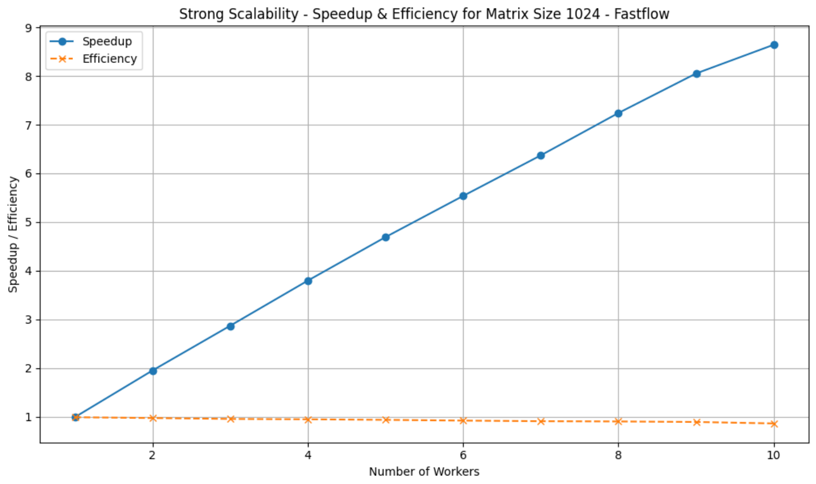

Figure 4: FastFlow speedup and efficiency under strong scaling: 1024 matrix size.

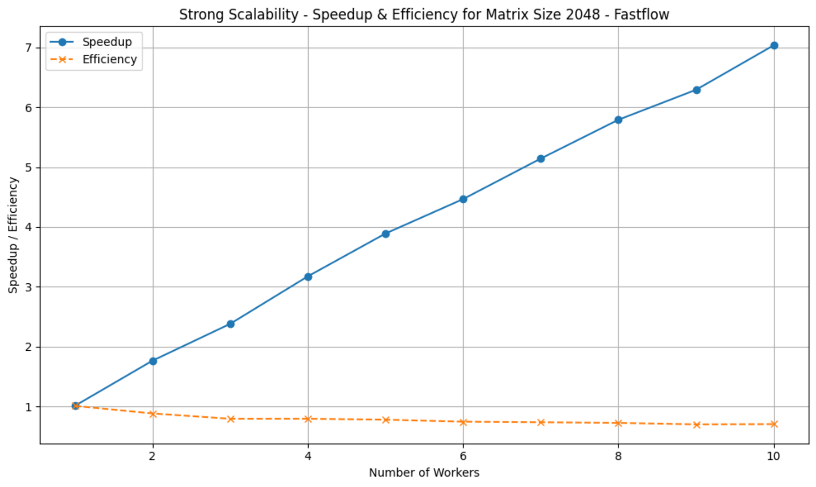

Figure 5: FastFlow speedup and efficiency under strong scaling: 2048 matrix size.

5.2.2 Weak Scaling

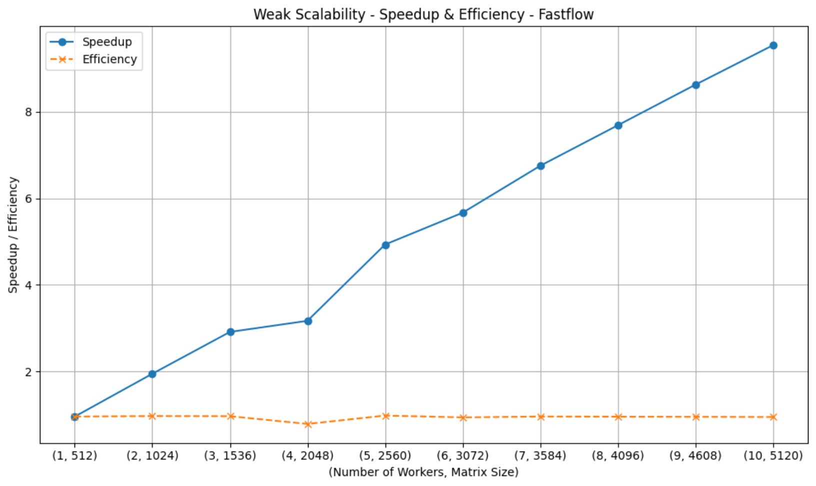

Figure 6: FastFlow speedup and efficiency under weak scaling: variable matrix size.

The plots indicate that the system exhibits good scalability, with the speedup increasing consistently as the number of workers grows. While the speedup is sublinear, it reflects significant performance improvements through parallelism. Efficiency declines slightly with the addition of workers, which is expected due to parallelization overheads such as communication and synchronization.

5.3 MPI Version

For the MPI version, the following configurations were tested:

- Number of processes: Scaling from 1 to the total number of available nodes in the *spmcluster* (7 nodes), i.e., $i \in \{1, \ldots, 7\}$.
- Matrix size: Scaled with the number of processes, following the formula $512 * i$, where $i \in \text{range}(1, \text{max\_num\_processes} + 1)$.

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

P\M	**512**	**1024**	**1536**	**2048**	**2560**	**3072**	**3584**
1	19	194	748	2115	4562	8950	14759
2	39	159	473	1190	2392	4438	7607
3	52	166	406	907	1761	3187	5238
4	60	161	367	755	1468	2543	4143
5	84	221	435	800	1428	2340	3669
6	81	193	371	683	1213	1953	3211
7	106	276	508	840	1385	2089	3086

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

Smaller matrices demonstrate suboptimal performance due to thread overhead outweighing computational benefits.

5.3.1 Strong Scaling

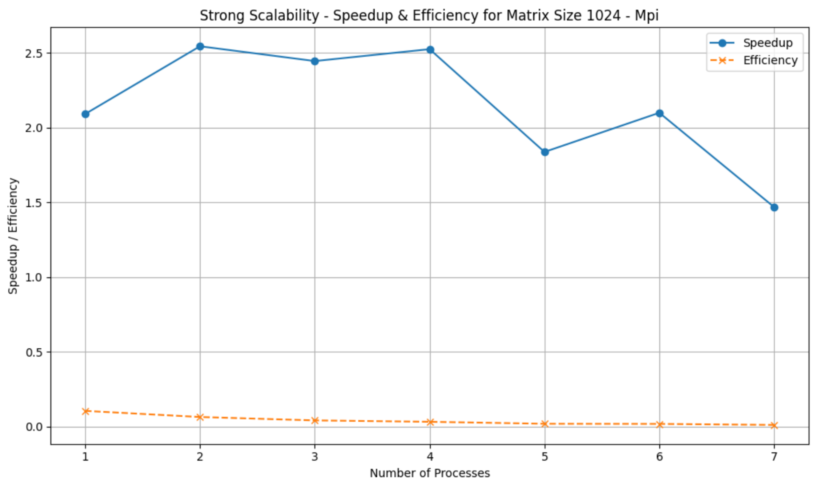

Figure 7: MPI speedup and efficiency under strong scaling: 1024 matrix size.

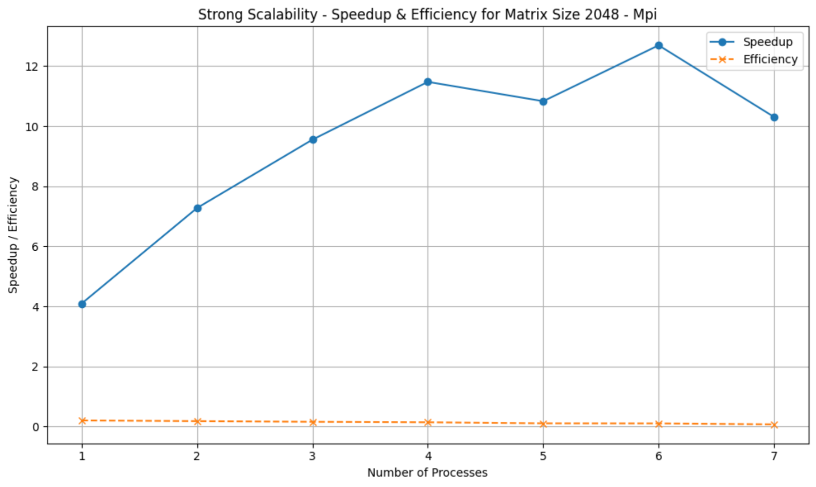

Figure 8: MPI speedup and efficiency under strong scaling: 2048 matrix size.

5.3.2 Weak Scaling

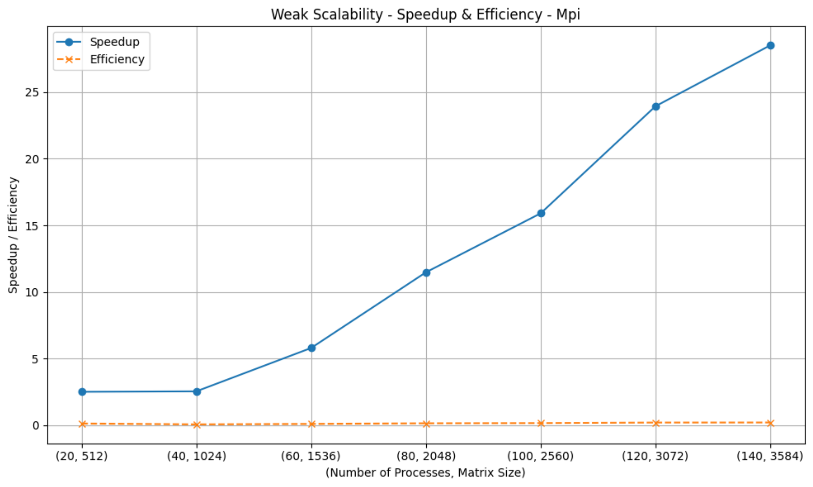

Figure 9: MPI speedup and efficiency under weak scaling: variable matrix size.

The MPI implementation demonstrates a high speedup for larger matrices, particularly under weak scaling. However, efficiency is notably lower than in the FastFlow.

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 sizes) for both the FastFlow and MPI versions.

6.1 FastFlow Version

Matrix Size	Workers	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
3584	10	9147.33	$\frac{T(1)}{T(p)} = \frac{87989.55}{9147.33} = 9.62$	$\frac{S(p)}{p} = \frac{9.62}{10} = 0.96$
4096	10	16520.11	$\frac{T(1)}{T(p)} = \frac{158846.02}{16520.11} = 9.62$	$\frac{S(p)}{p} = \frac{9.62}{10} = 0.96$
5120	10	33248.18	$\frac{T(1)}{T(p)} = \frac{316855.61}{33248.18} = 9.53$	$\frac{S(p)}{p} = \frac{9.53}{10} = 0.95$

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

6.2 MPI Version

Matrix Size	Processes	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
3584	7	3085.89	$\frac{T(1)}{T(p)} = \frac{87989.55}{3085.89} = 28.51$	$\frac{S(p)}{p} = \frac{28.51}{140} = 0.20$
3072	6	1952.63	$\frac{T(1)}{T(p)} = \frac{46748.92}{1952.63} = 23.94$	$\frac{S(p)}{p} = \frac{23.94}{120} = 0.20$
2560	6	1213.48	$\frac{T(1)}{T(p)} = \frac{22708.80}{1213.48} = 18.71$	$\frac{S(p)}{p} = \frac{18.71}{120} = 0.16$

Table 4: Top 3 Configurations with the Best Speedup and different Matrix Sizes for MPI-based Version

6.3 Analysis

As seen in the tables and through the plots, the MPI version performs quite well in terms of weak scalability, achieving a speedup of approximately 29x. However, it also exhibits significant inefficiency, as it utilizes up to 140 processing entities (20 threads per core with 7 processes, each assigned to one node), which results in a notably low efficiency of 0.20.

On the other hand, the FastFlow version demonstrates much more consistent performance, handling various matrix sizes effectively. The speedup is stable across different configurations, and the efficiency is significantly higher, with values around 0.95. This stability makes the FastFlow implementation a better choice for applications with varying workloads, as it delivers solid performance without wasting computational resources.