SPM Report

Leonardo Crociani - 615392

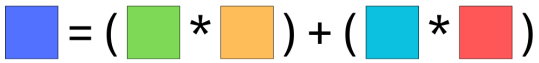
January 23, 2025

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:



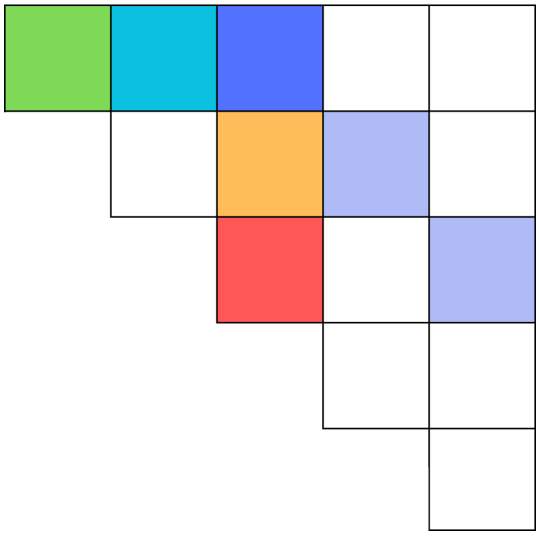


Figure 1: Wavefront single element computation

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` on an internal node using `srun`.
   (c) Build the code on an internal node also using `srun`.
   (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> <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 several minutes

4 Cache access optimization

The matrix is stored as an array of arrays, which in C++ results in a row-major memory layout. Since only the upper triangular part contains meaningful values, we can utilize the unused lower triangular portion to store the transposed values. This allows us to access the next required values more efficiently by taking advantage of the cache's spatial locality principle, ultimately speeding up computation.

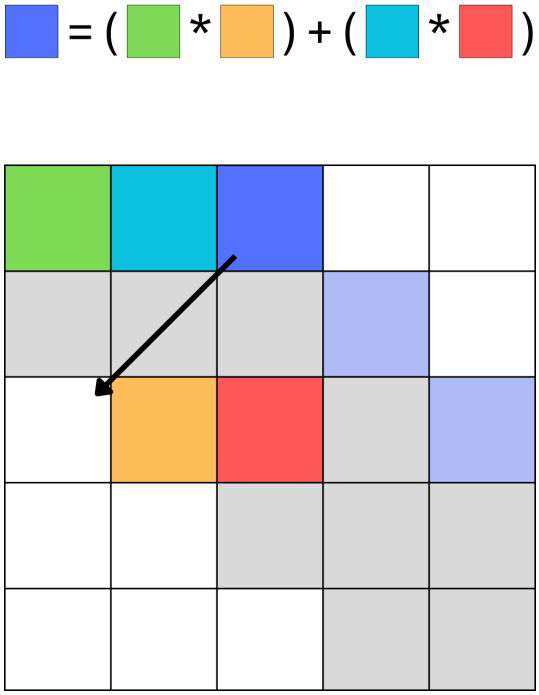


Figure 2: Wavefront single element computation with cache optimization. The computed value will be stored also in the location indicated by the arrow. Grey cells are already computed values.

5 Parallelization Strategy

5.1 FastFlow

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

5.1.1 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. In the meanwhile, the emitter do some work too.
- 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.

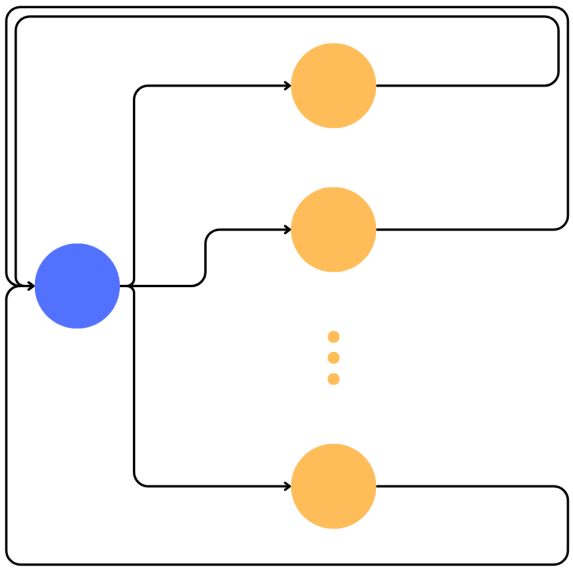


Figure 3: Farm schema. The blue node represents the emitter, while the other nodes are the workers.

5.1.2 Workload Balancing

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

5.2 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.

5.2.1 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.

5.2.2 Workload Balancing

The range of action for each process is computed using a static block distribution, similar to the FastFlow implementation. The communication in the final iteration is unnecessary and can be skipped. Furthermore, only the process with rank zero performs the computation in the last iteration, as it is also responsible for printing timing and the upper right value.

6 Performance Analysis and Plots

I chose to use two different matrix sizes: 256×256 for FastFlow and 512×512 for the MPI version. This decision is based on the underlying considerations:

- FastFlow operates on a shared memory system where memory is shared among threads, enabling rapid communication. Smaller matrices (256×256) are preferred because larger matrices lead to increased cache misses, which degrade performance. For example, doubling the matrix size from 4096 to 8192 resulted in approximately 13x more cache misses[6].
- MPI is designed for distributed memory systems, where processes communicate over a network. Using small matrices would result in communication overhead dominating the execution time, reducing the meaningfulness of the results. Larger matrices (512×512) ensure that computation dominates over communication, leading to more representative performance measurements.

Additionally, I decided to plot a speedup comparison under weak scaling conditions for both approaches, using an intermediate matrix size of 384.

6.1 Sequential Version

For the sequential version, I tested the implementation using matrix side sizes defined as $256 * i$ for $i \in \{2, 4, \ldots, 32\}$, to cover both FastFlow and MPI comparisons.

The execution time (measured in milliseconds[7]) follows an exponential trend, as shown in the plot below.

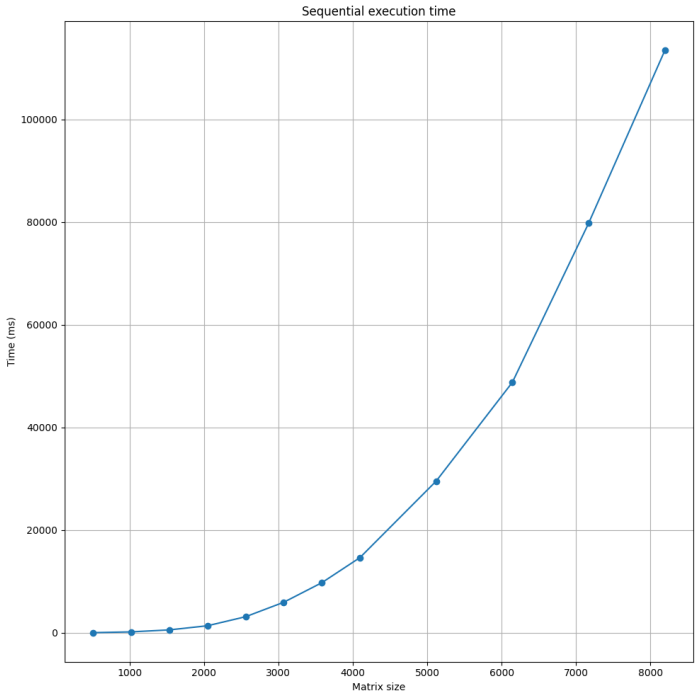


Figure 4: Execution times for the sequential version.

6.2 FastFlow Version

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

- Number of threads: This parameter varies from 2 to 16. Note that at each iteration the total number of workers is `num_threads` - 1, as one thread is dedicated to the emitter.

[6] Tested on the frontend node using the `perf` command.

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

- Matrix side size: Defined as $256 * i$, where $i \in \{2, 4, 6, \ldots, 16\}$.

Table 6.2 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	**512**	**1024**	**1536**	**2048**	**2560**	**3072**	**3584**	**4096**
2	12	87	272	693	1667	3301	5437	8214
4	7	45	142	361	936	1955	3289	4973
6	5	30	96	270	725	1639	2842	4340
8	4	24	72	191	644	1571	2773	4287
10	4	21	61	150	383	1087	2073	3273
12	5	19	52	117	258	704	1607	2739
14	5	17	47	101	204	509	1403	2464
16	5	18	44	94	179	470	1360	2311

Table 1: Execution times (in milliseconds) for various matrix side sizes and threads counts.

6.2.1 Strong Scaling

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

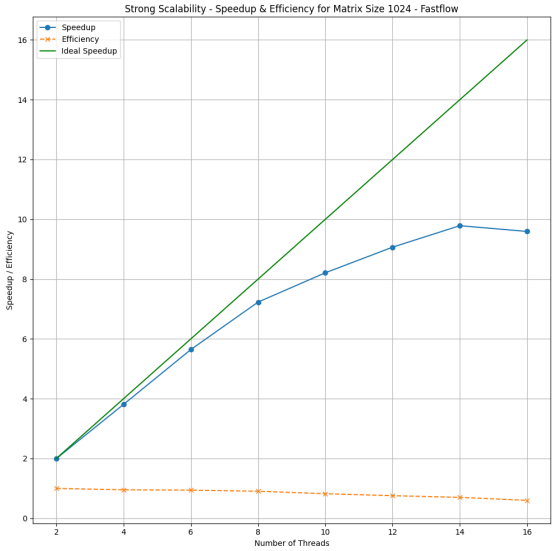


Figure 5: FastFlow-based version speedup and efficiency under strong scaling: 1024 matrix side size.

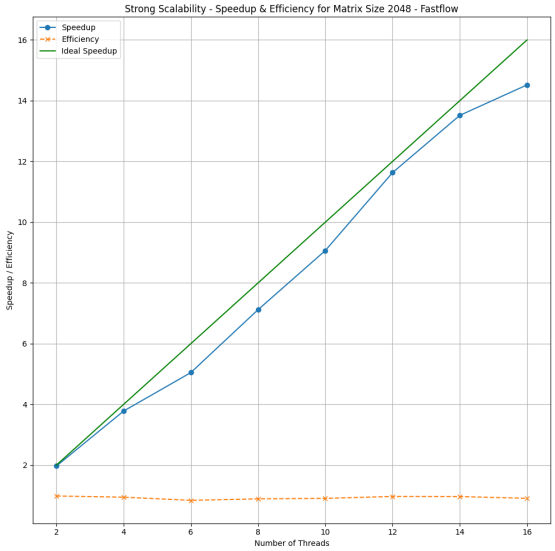


Figure 6: FastFlow-based version speedup and efficiency under strong scaling: 2048 matrix side size.

As expected, the solution scales better with larger matrices. Let us consider the 1024×1024 case. At a certain point, as the number of threads increases, the speedup begins to decrease. This is because the workload per thread becomes smaller, and the overhead associated with thread management and synchronization starts to dominate over the actual computation. In contrast, for the 2048×2048 matrix size, the higher computational workload per thread reduces the relative impact of overhead, allowing the solution to scale more effectively.

6.2.2 Weak Scaling

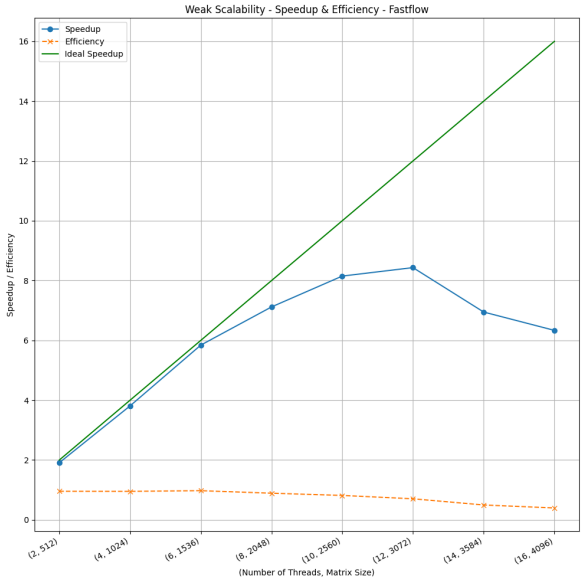


Figure 7: FastFlow-based version speedup and efficiency under weak scaling: variable matrix side size.

In terms of weak scaling (Fig. 7), we observe that beyond a certain matrix size, the speedup begins to start decreasing. This is likely because the cumulative cache is insufficient to effectively handle the larger matrix sizes. This is likely because the cumulative cache becomes insufficient to handle the larger matrix sizes effectively. As mentioned earlier, doubling the matrix size significantly increases the number of cache misses due to the limited cache capacity and the resulting memory access overhead, which negatively impacts performance.

6.3 MPI Version

For the MPI implementation, the following configurations were tested:

- Number of nodes: tested the range {1, 2, 4, 8}.
- Number of processes: the number of processes used was $i \in \{1, 2, 4, 6, \ldots, 16\}$.
- 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., $i \in \{1, 2, 4, \ldots, 16\}$.

The table below presents the execution times for different combinations of processes (**P**) and matrix side sizes (**M**) considering only 2 processes per nodes:

P\M	**512**	**1024**	**2048**	**3072**	**4096**	**5120**	**6144**	**7168**	**8192**
1	25	169	1318	5848	14318	29280	47995	79046	111209
2	36	143	826	2623	7334	15001	26619	41476	60984
4	56	150	587	1616	3599	7386	14246	22306	33270
8	78	204	602	1266	2369	4012	7039	12019	18248

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

6.3.1 Strong Scaling

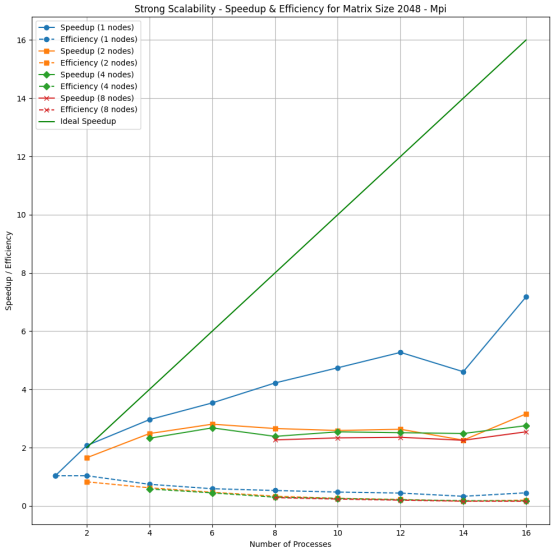


Figure 8: MPI-based version speedup and efficiency under strong scaling: 2048 matrix side size.

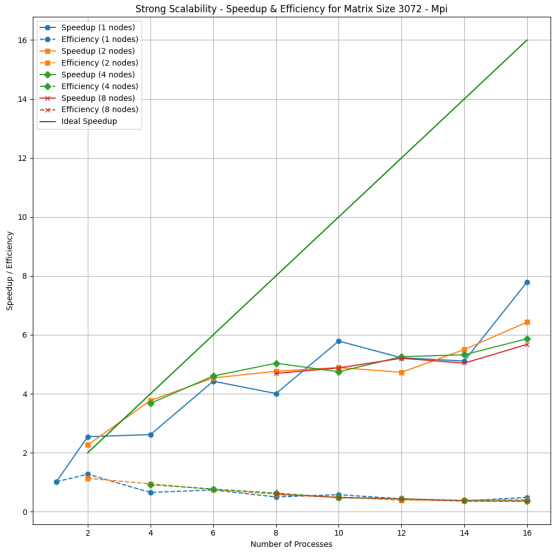


Figure 9: MPI-based version speedup and efficiency under strong scaling: 3072 matrix side size.

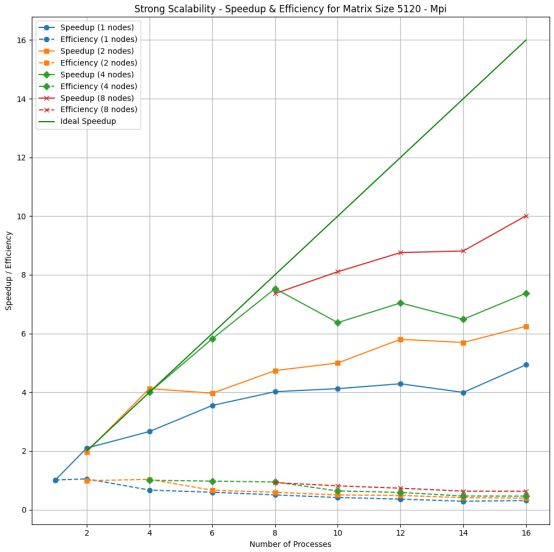


Figure 10: MPI-based version speedup and efficiency under strong scaling: 5120 matrix side size.

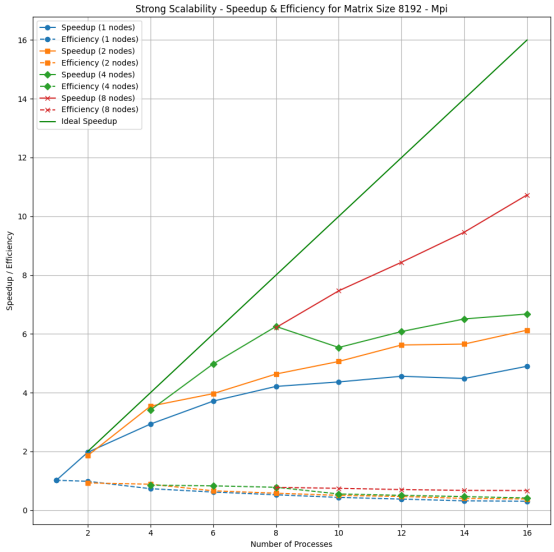


Figure 11: MPI-based version speedup and efficiency under strong scaling: 8192 matrix side size.

When the matrix size is small (e.g., 2048 – Fig. 8), the optimal approach is to allocate all processes to a single node. As the matrix size increases, distributing the processes across multiple nodes becomes more efficient. Initially, the cost of communication dominates over computation. Therefore, on a single node, where communication is faster, better speedups can be achieved. For a matrix size of 3072, the costs of communication and computation are approximately balanced. With a matrix size of 8192, leveraging distinct memories and cache levels by distributing processes across the cluster nodes becomes the optimal solution. In this case, we achieved a speedup of nearly 11x.

6.3.2 Weak Scaling

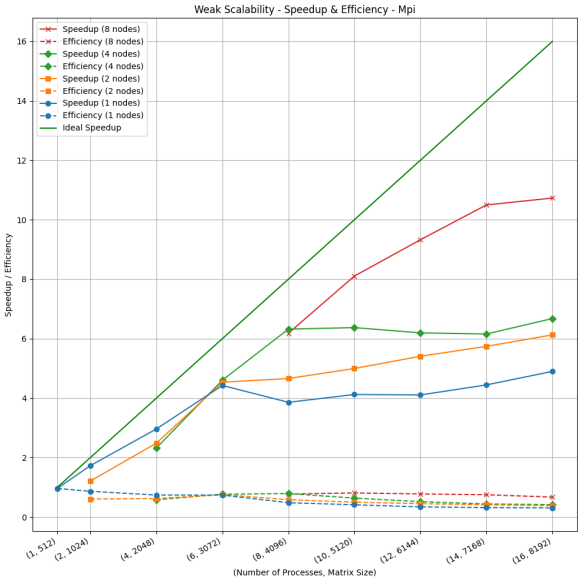


Figure 12: MPI-based version speedup and efficiency under weak scaling: variable matrix side size.

6.4 Fastflow and MPI

Here's the plot of both, in presence of weak scaling, when considering a matrix that is in the middle between the other two: size is 384.

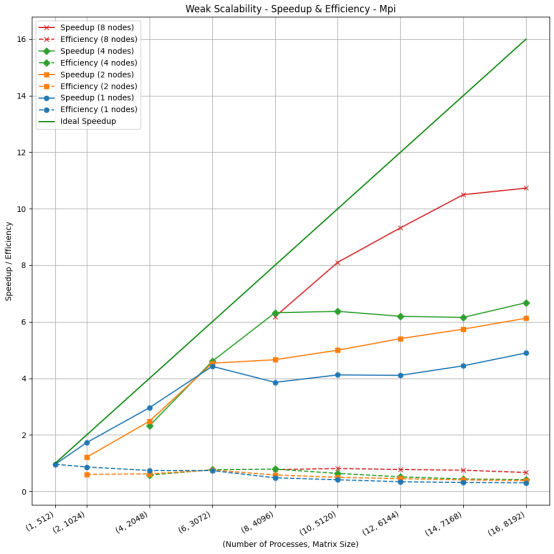


Figure 13: MPI-based version speedup and efficiency under weak scaling with matrix size equal to 384 * `num_processses`.

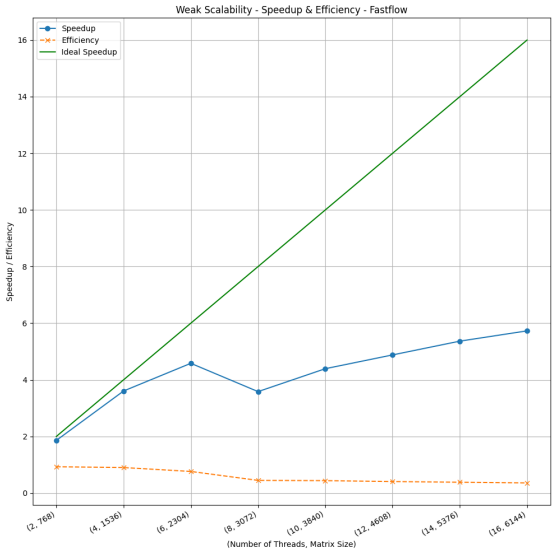


Figure 14: FastFlow-based version speedup and efficiency under weak scaling with matrix size equal to 384 * `num_threads`.

7 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.

7.1 FastFlow Version

Matrix side Size	Threads	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
2560	16	179.46	$\frac{3125.32}{179.46} = 17.42$	$\frac{S(p)}{p} = \frac{17.42}{16} = 1.09$
2048	16	94.04	$\frac{1365.46}{94.04} = 14.52$	$\frac{S(p)}{p} = \frac{14.52}{16} = 0.91$
3072	16	470.75	$\frac{5941.55}{470.75} = 12.62$	$\frac{S(p)}{p} = \frac{12.62}{16} = 0.79$

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

7.2 MPI Version

Matrix side Size	Processes	Time $T(p)$	Speedup $S(p)$	Efficiency $E(p)$
7168	16	6660.00	$\frac{79769.48}{6660.00} = 11.98$	$\frac{S(p)}{p} = \frac{11.98}{16} = 0.75$
6144	16	4542.00	$\frac{48791.81}{4542.00} = 10.74$	$\frac{S(p)}{p} = \frac{10.74}{16} = 0.67$
5120	16	2950.00	$\frac{29540.14}{2950.00} = 10.01$	$\frac{S(p)}{p} = \frac{10.01}{16} = 0.63$

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

7.3 Analysis

As expected, larger matrix sizes have a significant impact on both speedup and efficiency for the two implementations. On one hand, FastFlow performs better with smaller matrices due to the fast communication between threads within a single node. On the other hand, MPI scales well with larger matrices by efficiently distributing processes across multiple nodes.