UNIVERSITÀ DI PISA

Department of Computer Science
Master Degree in Artificial Intelligence

SPM PROJECT: Distributed Wavefront computation

Author:

Francesco Simonetti

A.Y. 2023/24

Contents

Chapter 1

Introduction

1.1 Problem description

Distributed Wavefront computation Consider the same problem of Assignment 1 (wavefront computation). For each diagonal element of the matrix M (NxN) of double precision elements, the distributed version computes the $n - k$ diagonal element $e^k_{m,m+k}$ ($m[0, n - k[$) as the result of a dotproduct operation between two vectors v^k_m and v^k_{m+k} of size k composed by the elements on the same row m and on the same column $m + k$. Specifically

$$e^k_{i,j} = \sqrt{\text{dotprod}(v^k_m, v^k_{m+k})}$$

The values of the element on the major diagonal $e^0_{m,m}$ are initialized with the values $(m + 1)/n$. Two parallel versions have been implemented for solving the problem:

1. For a single multi-core machine using the FastFlow library
2. For a cluster of multi-core machines using MPI

Chapter 2

Solution methods

Before delving into the specific methods, the general logic used to parallelize the problem will be presented, as it will be applied in both versions of the solution.

2.1 General work distribution Logic

The computation of each diagonal depends on the computation of the previous diagonals. Therefore, to calculate the k-th diagonal, all diagonals from 0 to k - 1 must have already been calculated. This dependency makes this part of the problem non-parallelizable, and thus, the diagonals will be calculated sequentially, one at a time. Conversely, the computation of a specific diagonal can be parallelized. Indeed, for a given diagonal k, the computation of element i is independent of the computation of element j for every i and j within the diagonal k.

When parallelizing the computation of diagonals with a fixed number of workers (wich could be thread or processes) and with diagonals becoming progressively smaller, the distribution process works as follows:

1. **Chunk Division**: For each diagonal to be computed, the total number of elements is divided by the number of available workers. Each worker will have a chunk.of this division, along with the indices indicating the section of the diagonal to work on. For example, for a diagonal of size 12 with 3 workers, the chunk size will be 4 elements: worker 1 will have `start = 0` and `end = 4`, worker 2 will have `start = 4` and `end = 8`, and so on. `start` and `end` indices define the section of the diagonal to be processed.

2. **Handling Remainders**: When the total number of elements is not exactly divisible by the number of workers, the chunk size for each worker is calculated as the largest integer less than or equal to the result of dividing the number of elements by the number of workers. The remaining elements, which cannot be evenly distributed, are assigned to the last worker. For example, if a diagonal has 7 elements and there are 3 workers, each worker receives $\left\lfloor \frac{7}{3} \right\rfloor = 2$ elements, and the last worker receives the remaining 1 element.

3. **Case of Fewer Elements than Workers**: If the size of the diagonal is less than the number of workers, each element is assigned to a worker. Therefore, in these cases, only certain workers perform actual work by computing a portion of the diagonal.

2.2 FastFlow-Based Solution

The code consists of the following main components:

1. **Main**
2. **Workers**
3. **Emitter**

Within the `main` function, a worker farm is created, with the number of workers specified by a parameter passed via the command line. Additionally, an $n \times n$ matrix is allocated, also determined by a command line parameter. After initialization, `main` starts the `Emitter`, passing the matrix and the worker farm as arguments, thus beginning the parallel computation of the matrix.

Emitter The Emitter is responsible for coordinating the work between workers for computing the diagonals of the matrix. When it comes to computing the k-th diagonal, the Emitter divides the work into chunks and assigns tasks to the available workers, passing them the matrix and indices needed to identify the section of the diagonal to be computed (as shown in 2.1).

The Emitter must start the workers for computing diagonal k only when the computation of diagonal $k-1$ has been completed, as there is a dependency between diagonals. To manage this synchronization, a feedback-based mechanism is used.

Feedback Mechanism

Consider the case where diagonal $k-1$ has just been computed (as illustrated in Figure 2.1). The Emitter, after creating the chunks, sends the tasks to the workers, and during this process, it keeps track of the number of tasks sent using a counter called `expected_feedback`. This counter is used because in the final diagonals, the number of threads that will be executed will differ from the number of available workers. At the same time, the Emitter maintains another counter, `feedback_count`, which tracks the number of feedbacks received from the workers.

When a worker finishes its task, it sends an empty feedback to the Emitter to signal the completion of the work. Upon receiving a feedback, the Emitter executes its `svc` function, which handles various checks:

- If a feedback has been received from a worker, the `feedback_count` counter is incremented.
- If `feedback_count` equals `expected_feedback`, it means that all workers have completed the computation of the current diagonal. At this point, the Emitter

increments the diagonal index (`current_k`) and repeats the entire process of task division and worker dispatching for the computation of the new diagonal.

- If `feedback_count` has not yet reached `expected_feedback`, the Emitter simply waits for the remaining feedbacks to arrive without taking further actions.

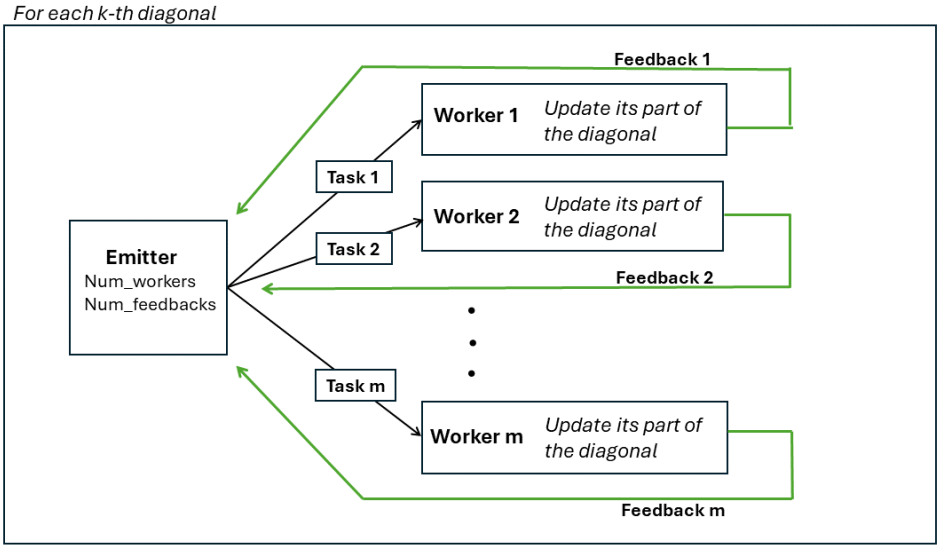


Figure 2.1: Emitter - Workers logic

Observation: When the Emitter is launched, it initially receives a feedback signal from `main`. Therefore, the Emitter must begin execution to calculate the first diagonal without having received any feedback from the workers, but only the initial one from `main`. This is managed with a simple check that verifies whether the feedback comes from `main` (which is identified by nullptr).

2.3 MPI-Based Solution

In MPI, each worker corresponds to an independent process that performs a portion of the diagonal computation of a matrix. Unlike FastFlow, MPI does not have a central coordinator: each process independently calculates its own part. Since processes in MPI do not share memory, the chosen approach is to provide each process with a complete copy of the matrix. Although this approach is memory-intensive, it avoids the need for continuous data exchanges between processes, simplifying the code logic.

First, the chunk size is calculated for each process. Then, based on its rank and the chunk size, each process can determine which sections of the diagonal it needs to compute, following the approach explained in 2.1. In this case, the process with rank 0 will compute the first part of the diagonal, whose size is determined by the

calculated chunk, the process with rank 1 will compute the second part, and so on. After each process calculates its portion of the diagonal, all partial results are collected and distributed among the processes through a gathering operation.

Specifically, the gathering operation allows each process to send its results and receive results computed by others. Thus, at the end of each iteration, all processes have a complete copy of the updated diagonal, which they can use for subsequent iterations (as in figure 2.2).

A crucial aspect is that the gathering operation blocks the outer loop iterating over the diagonals until all processes have completed their work and sent their results. This ensures that no process proceeds with calculating the next diagonal until all are synchronized on the same portion of the matrix.

Another important consideration is when the number of elements to compute on the diagonal becomes smaller than the number of available processes. In this situation, some processes may have no elements to process. Since every process is still involved in the collective communication (gathering), even those with no work in a particular iteration send a count of zero elements but still participate in the gathering, receiving data computed by other processes. This ensures proper synchronization, and processes can continue with the calculation of the next diagonal.

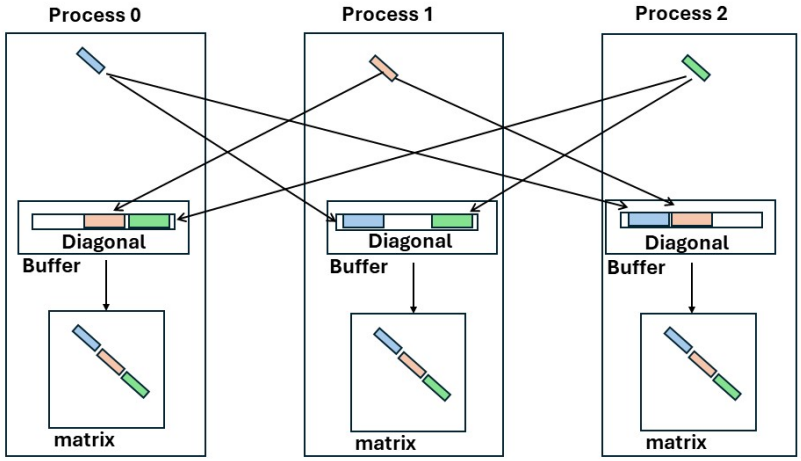


Figure 2.2: MPI Logic

Chapter 3

Test

OSS: For simplicity, whenever the term "matrix size" is used in this context, it will refer to the number of rows or, equivalently, the number of columns, as the matrices in this problem are square.
OSS. For weak scaling tests, it is necessary to increase the number of threads/processes as the matrix size grows to ensure that the number of elements each thread/process works on remains constant. To maintain this constant ratio, a ratio value is used, which must remain unchanged. The ratio is calculated as follows:

weak ratio = matrix size / number of workers.

3.1 Fastflow test

The scalability test on a single thread was not performed, as the FastFlow code requires at least two threads to function: an emitter and a worker. Initially, it was considered to run the test using two threads, with the idea that only one of them (the worker) would actually perform the diagonal calculations. However, this approach is not entirely accurate, as while the worker is busy calculating the diagonal, the emitter simultaneously calculates the chunk sizes and the start and end indices of the diagonal to be passed to the worker for the next iteration.

OSS. for the weak scaling tests, the weak ratio was calculated using the number of workers, excluding the emitter. This approach ensures that the workload assigned to each thread workers remains consistent as the matrix size increases. However, in the actual computation of the matrix, all threads, including the emitter, were considered. For example, to calculate weak efficiency with a weak ratio of 100 and a matrix size of 400, 4 workers would be used because $\frac{400}{4} = 100$, and the efficiency formula would be:

$$\text{Efficiency} = \frac{\text{Sequential Time}}{\text{Parallel Time (using 5 threads: 4 workers + 1 emitter)} \times 5}$$

Similarly, for a matrix size of 500, 6 threads (5 workers + 1 emitter) would be used, and for a matrix size of 800, 9 threads, and so on.

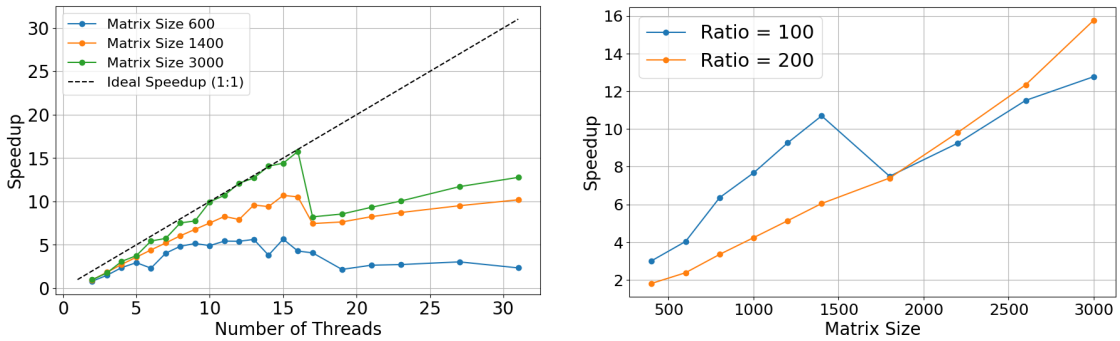
3.1.1 Fastflow Speedup, strong and weak

Strong Speedup figure 3.1 a
This test analyzes the strong speedup for different fixed matrix sizes: 600, 1400, 3000. It is observed that up to around 15 workers (16 threads), the speedup increases for all three matrices, particularly for the largest matrix (3000), the ideal speedup is achieved for various combinations before reaching 16 threads.

However, beyond the threshold of 16 threads, a significant drop in performance occurs. For the larger matrices (1400 and 3000), after this "drop," there is a gradual increase in speedup again, resulting in a double growth. This behavior can be explained by the number of physical cores available on a server node, which is 16. Once this limit is passed, additional overhead is introduced due to thread switching during scheduling. This overhead is gradually slightly amortized (for 3000 and 1400) but never reaches the performance obtained with 16 threads. It is likely that the logical cores are not as efficient in handling the specific workload of this problem.

In contrast, for smaller matrices (600), after the "drop" at 16 threads, there is no second increase in speedup. In this case, the overhead caused by the large number of threads for the limited amount of computation is not sufficiently amortized.

Weak Speedup figure 3.1 b
The weak speedup test was conducted with two different **weak ratios** (second observation in 3): 100 and 200. It was noted that for smaller matrices, a lower ratio of elements per thread favors weak speedup, as more workers are actively engaged in computation. However, as the matrix size increases, and consequently the number of threads, better results are obtained with a higher ratio (200). This is consistent with the observations in strong speedup, where excessive use of threads, especially more than 15, tends to degrade performance.

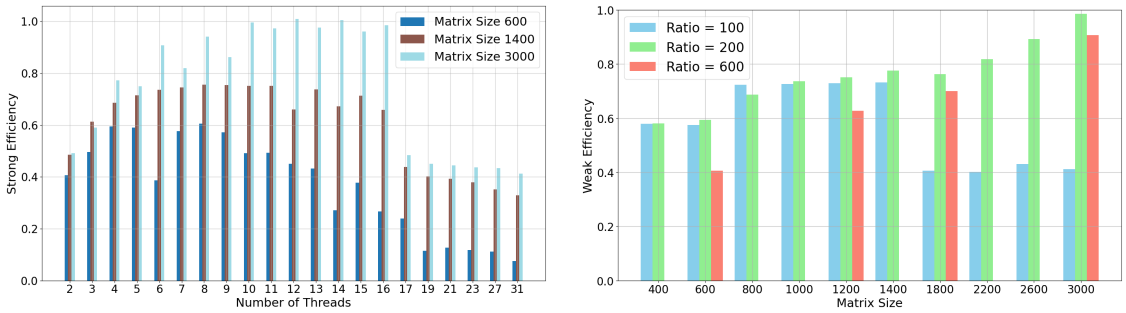


(a) *Strong Speedup* (b) *Weak Speedup*

Figure 3.1: Speedup

3.1.2 Fastflow Efficiency: Strong and Weak

Strong efficiecny figure 3.2 a
Better efficiency is achieved with large matrices; in fact, only with matrices of size 3000 does efficiency approach 1 in certain intervals and thanks to the caching effect for some combinations achieve it greater than one. For smaller matrices, efficiency tends to remain very low and it gets worse as the number of threads increases. For medium to large matrices, confirming what was observed in the speedup, there is a significant drop in performance around 15 threads. In all matrices, with a low number of threads, efficiency is poor because, proportionally, the number of workers relative to the total number of threads is low, considering that there is always an emitter that solely handles work distribution.
Weak efficiecny figure 3.2 b
A low ratio (100) shows reduced efficiency, which further deteriorates for matrices larger than 1500 due to the use of more than 15 threads. For a medium ratio (200), efficiency is consistently better compared to a ratio of 100, with no observed decline as the matrix size and number of threads increase, since the critical value of more than 16 threads is never reached. For a high ratio (600), the trend is similar to that of a ratio of 200, but with slightly lower efficiency and a more pronounced progression. It is likely that there is an intermediate ratio between the two that would yield better overall efficiency.



(a) *Strong Efficiency* (b) *Weak Efficiency*

Figure 3.2: Efficiency

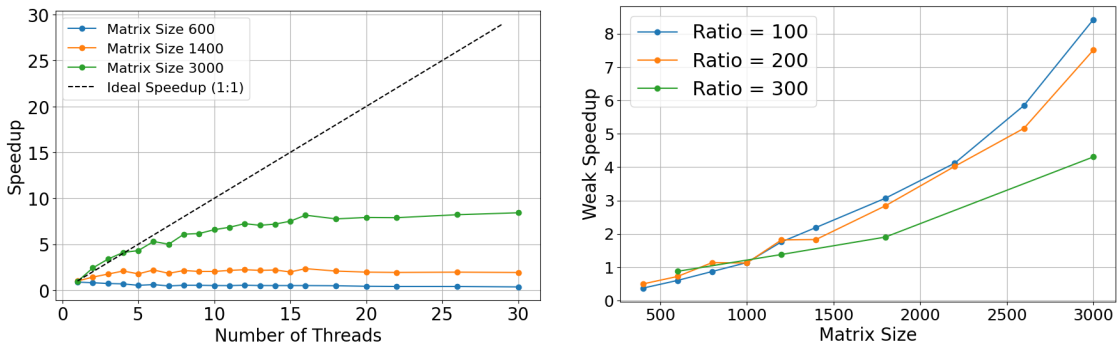
3.2 MPI test

Due to technical constraints and server congestion, MPI tests have only been conducted using a maximum of 5 nodes. Consequently, all results presented are based on tests performed with up to 5 nodes, with the exception of cases involving 1 to 4 processes where the number of nodes matches the number of processes.
During the development of the code, numerous "interim" tests were conducted. As expected, using fewer nodes often yielded better performance for a relatively small number of processes. For example, with fewer than 16 processes, performance was consistently better with two nodes compared to one. However, this trend reversed

with more than 16 processes. This was evident from the speedup tests of fastflow, which showed that each node performs optimally with up to 16 threads or processes. Therefore, as long as the number of processes per node is limited to 16, performance improves with fewer nodes.
This observation was verified during development but unfortunately could not be formalized with an actual plot due to a lack of additional testing.

3.2.1 MPI Speedup, strong and weak

Strong Speedup figure 3.3.a
Analyzing the speedup calculated with three different matrices (600, 1400, 3000), we can observe that there are no significant drops in performance for any of them. For small matrices, the speedup decreases slowly because the communication cost between processes is high and is not amortized by the limited amount of computation per process. For medium-sized matrices (1400), a similar situation is observed, but without any significant improvement; instead, the speedup remains relatively constant. For large matrices (3000), the speedup consistently improves and, in some configurations, even achieves super-linear speedup,such as with 3 processes, where the runtime improves by more than three times compared to the sequential execution. However, as the number of threads increases, the rate of speedup improvement declines, as the communication overhead becomes more costly, and an even larger matrix would be needed to offset the high number of processes.
Weak Speedup figure 3.3.b
The weak speedup calculated with three different ratios shows a similar upward trend across all cases. For ratios 100 and 200, the speedup obtained is very close, with the 200 ratio performing slightly worse, but only marginally considering that it uses half the processes compared to the 100 ratio. With a very high ratio of 600, as expected, the speedup is lower, but since it uses six times fewer processes than the 100 ratio and three times fewer than the 200 ratio, the resulting speedup remains relatively high compared to the other two cases.

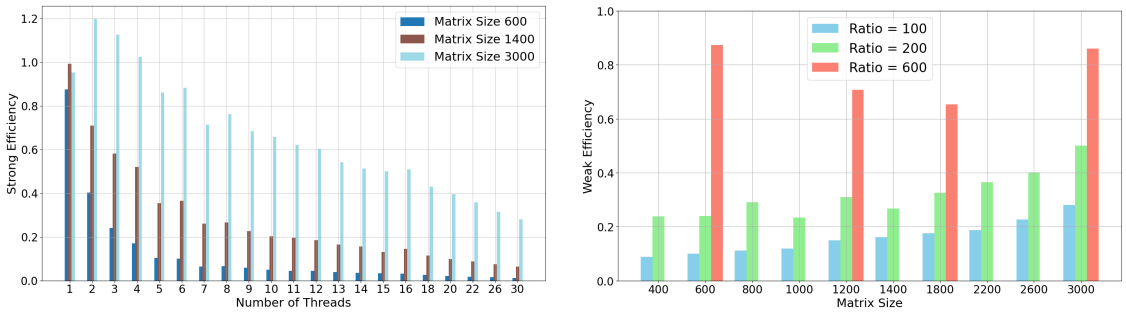


(a) *Strong Speedup* (b) *Weak Speedup*

Figure 3.3: Speedup

3.2.2 MPI Efficiency, strong and weak

Strong Efficiency 3.4.a
A confirmed by the speedup graph, only with a large matrix (3000) is a super-linear speedup achieved, meaning an efficiency greater than 1 (specifically with 2, 3, and 4 processes). These results are significantly better compared to those obtained by increasing the number of processes, not only because there is less communication between processes, but also for another reason: fewer nodes are used. As mentioned earlier, a maximum of 5 nodes are utilized, but when the number of processes is lower, the number of nodes also decreases accordingly. This results in fewer communications between nodes, which are much more expensive than communication between processes on the same node. In fact, even with smaller matrices, executions with a low number of processes show significantly better performance compared to any execution involving more than 5 processes.
Weak Efficiency 3.4.b
In the plot showing three ratios (100, 200, 600), we can observe that for lower ratios (100 and 200), there is a slight progressive increase but still poor efficiency. However, the trend is different with the higher ratio (600), which shows generally better results. Efficiency is initially high because, using a single processor (matrix size 600 means 1 process with ratio=600), there is minimal message passing overhead. After that, there is a decline in efficiency as the number of processors (2 and 3) increases, leading to more nodes and thus higher message passing costs. However, staying with the 600 ratio and moving to the last column with a very large matrix (3000), we see an improvement in efficiency again. This happens because, although the number of processors has increased, the number of nodes has not increased further, creating a "favorable" configuration for achieving good efficiency.



(a) *Strong Speedup* (b) *Weak Speedup*

Figure 3.4: Speedup

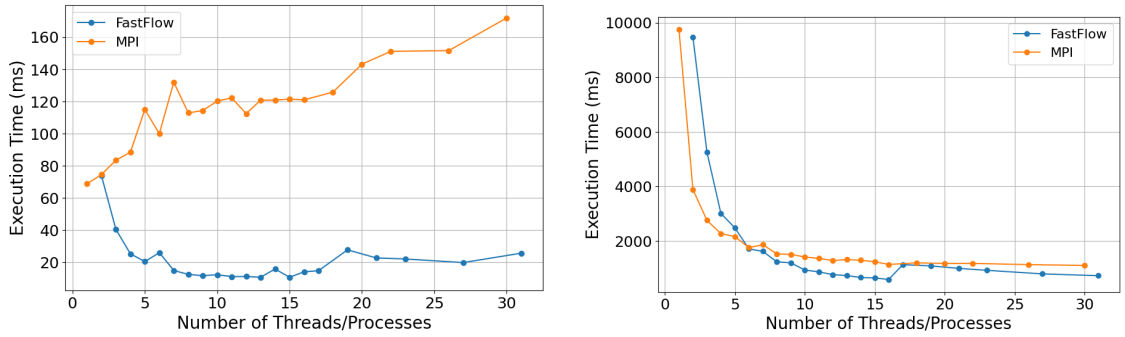
3.2.3 MPI Scalability

Salability plots were created since it is possible to run the program on a single process. However, I have chosen not to include them as they are not very significant. When using MPI with a single process, there is no real message passing, resulting

in execution times that are practically identical to those of the sequential version. Consequently, the scalability plots are almost identical to the speedup plots.

3.3 MPI and Fastflow observation

Before moving on to the observations comparing the various plots obtained, a graph will be presented that illustrates the difference in execution time between FastFlow and MPI for a given matrix size as the number of threads/processes increases. The plot on the left represents a matrix size of 1200, while the one on the right represents a matrix size of 3000.
The plots show that for smaller matrices, MPI's performance is significantly worse, regardless of the number of processes used. This is because the high cost of message passing between processes causes a rapid decline in performance. FastFlow, on the other hand, initially improves performance but then stabilizes, as the overhead from creating and managing threads limits any further gains.
For larger matrices, however, MPI better offsets the cost of communication between processes compared to FastFlow, especially when using a small number of threads or processes. This is because MPI utilizes all processes for computation, while FastFlow always dedicates one thread as an emitter, exclusively responsible for distributing the workload. As a result, FastFlow performs significantly worse with fewer threads or processes. As the number of threads increases, FastFlow eventually compensates for the overhead of the emitter, and both systems exhibit similar performance, converging around 16 threads/workers. At that point, FastFlow tends to experience slight performance degradation for reasons already discussed in the FastFlow testing section.



(a) *Execution time with matrix size 600* (b) *Execution time with matrix size 3000*

Figure 3.5: Fastflow and MPI execution time

3.3.1 MPI and fastflow speedup

We can clearly see that FastFlow, in several combinations (with fewer than 16 threads), gets close to the ideal speedup. MPI, on the other hand, has only a

few good combinations but generally shows lower speedup compared to the ideal and to FastFlow.

3.3.2 MPI and fastflow efficiency

Regarding efficiency, we immediately notice worse performance for MPI compared to FastFlow. However, an interesting observation is that as the ratio increases, MPI consistently improves, unlike FastFlow. In fact, with a ratio of 600 and a matrix size of 3000, FastFlow performs worse than with a ratio of 200, which is the opposite of what happens with MPI.

3.4 Consideration

From the tests, it would be very useful to run new experiments to see how MPI performs with much larger matrices, as it seems to work better with them. Another interesting thing would be to test MPI on multiple nodes to see how it behaves in that scenario.