UNIVERSITÀ DI PISA

Master Degree in Computer Science

Parallel and Distributed Systems: Paradigm and Models

Final project report

Francesco Botrugno

Master Degree in Computer Science

Contents

1 Introduction

This report presents the implementation of Project 3, proposed at the end of the SPM course (2023/2024). The objective is to compute LSH Similarity between objects. Starting from the sequential solution, I explored various parallelization approaches, addressing challenges and bottlenecks while striving to achieve the highest possible speedup. I implemented and analyzed solutions based on both the FastFlow (FF) paradigm and the MPI model, evaluating their efficiency and performance.

The first thing I did was decomposing the sequential algorithm into phases:

1. Initialize
2. Read from input file
3. Parse lines and generate elements
4. Compute the hash functions for each element
5. Build the hash map adding elements depending on their hash functions
6. Compare the collided elements and count similarities

To simplify the problem, I ignored the print on the optional output file. The critical stage of a parallel version of this program is between point 5 and 6, and in particular point 5 must be managed accurately to not lose similarities when operating on the map.

The time spent reading the input file has been excluded from the total computation time, the file is read and stored at the beginning, and the timer starts immediately after storing. This applies to the sequential version as well as the FF and MPI versions.

2 Fastflow version

The first idea I came up with was a pipeline of two farms, with an emitter that distributes file lines to the first farm, which parses them, generates elements, computes hash functions, and then adds them concurrently to a shared map. When all the elements have been inserted, a second farm operates reading the shared map and calculating similarities (Figure 1).

But this solution lead to a bottleneck in the concurrent access to the map and also needed a synchronization before the similarities computation, which adds more time to the algorithm.

Since the shared map caused a lot of overhead, I decided to create a local map for each worker of the first farm and merge them in the collector at the end. But this approach had the same problem as the previous solution: a bottleneck. The next idea was to calculate the similarities locally for each node and then sum the results together. The main challenge of this approach is that, to count the right number of similarities, all the element must be present, otherwise there is the risk of losing comparison. To have a significant local result, we need to split the map in independent sections.

The first split idea was to divide even and odd hash results, so I used as a second farm two nodes: one which operates on even hash functions and one on the odd ones.

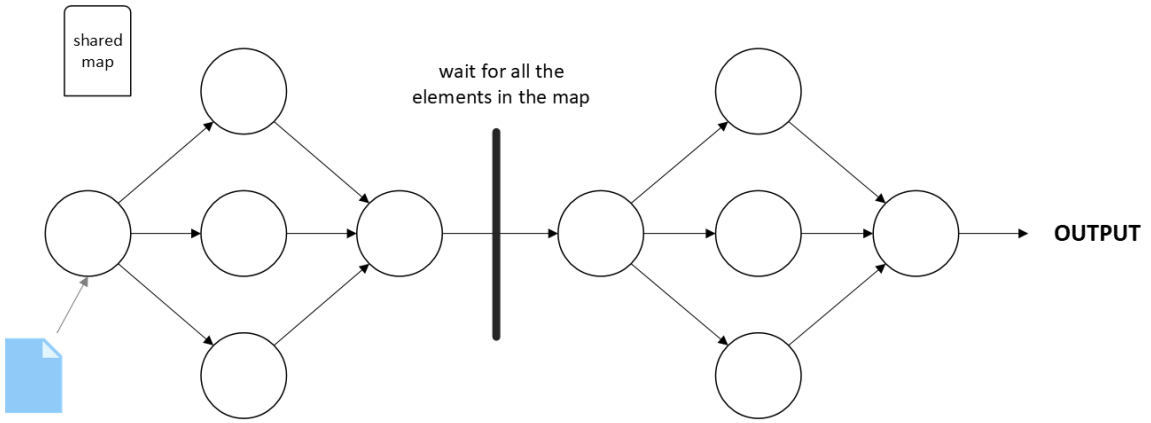

Figure 1: First solution with two farm, shared map and sync before accessing the second farm

This solution was better and did not need any barrier. But in this way if the number of workers of the first farm increases the workload for the two nodes increases too, leading to a bottleneck. So I generalized it extending the idea to n workers; instead of even and odd I decided to divide and assign the map based on the modulo operation h % nWorkers2 (Figure 2). This should balance the work if we use good hash functions, which means, hash functions that distribute the elements evenly.

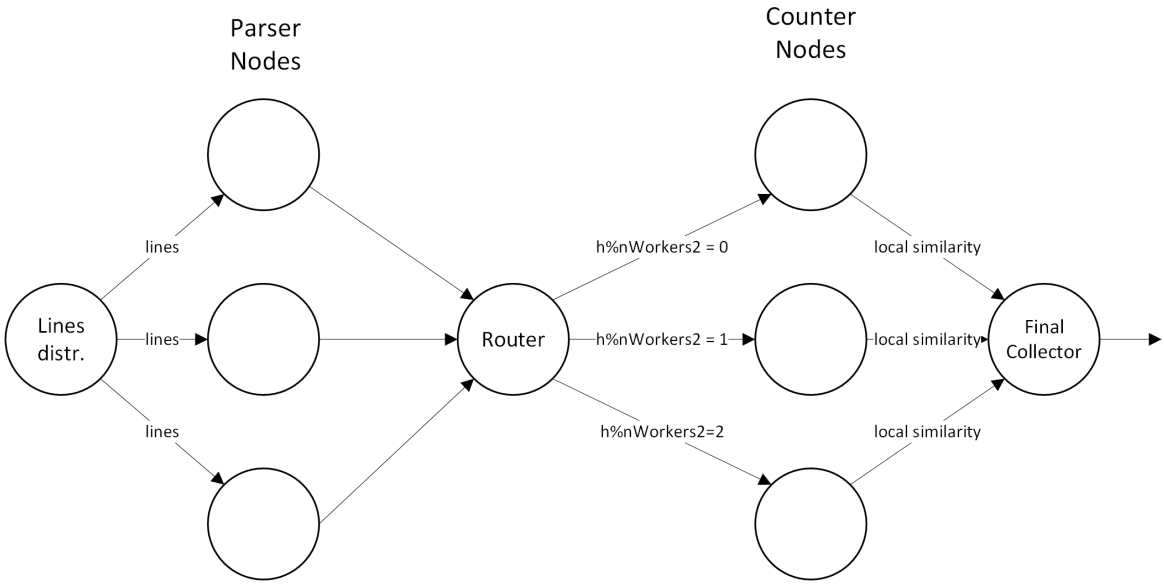

Figure 2: Final solution for the fast flow version with 2 farms of, possible, different size. Pipe(Farm1, Farm2)

After the set up of the model, I noticed that assigning lines one by one was generating too much overhead in the first stage, probably due to overhead communication between emitter and workers. I decided to assign lines in batches to each node in a (pseudo) round robin way. Choosing the optimal value of the batch size is a difficult task: it depends on the size of the input file and on the number of workers. I tried to find it empirically by launching different tests and I chose batch size equal to 1000, but for a better speedup, it is better to find the optimal batch size in a more precise way. The

final solution, which is implemented in the `fastflow.cpp` file is the following:

$$\text{Pipeline}(Farm1, Farm2)$$

with the possibility to choose two different sizes for the two farms. The completion time of this model is:

$$\text{Time} = Max(Farm1, Farm2)$$

1. **Emitter1 (Lines distributor)**: assigns batch of lines (pseudo) round robin to workers
2. **Workers1 (Parser)**: each worker of the first stage parses all the lines, collects the elements and sends them in chunks to the router
3. **Emitter2 (Router)**: routes the elements to the right worker based on the modulo operation `h%nWorkers2`
4. **Workers2 (Counter)**: adds the elements to its local map and calculates similarities, which are eventually sent to the final collector
5. **Collector (FinalCollector)**: sums all the received similarities and ends

Termination: Emitter1 sends EOS when reaches the end of the file, the EOS signal must not be propagated to the second farm. The final collector terminates only when receives all the final local similarities from the second farm.

3 MPI version

For the MPI version of the program I implemented two different algorithms: one which resembles the FF version and a second one which uses All-to-all communication.

3.1 Solution 1

The main node (Node 0) reads the file and stores it, then sends the lines to nWorkers of a first group in a round robin way. Each of these nodes parses the line and based on the module operation h % 2 forwards the element to the proper node of the second group (even or odd). I chose to use only even and odd since we have a maximum of 8 nodes, so 2 is a good proportion. The Even and Odd nodes, as in the FF solution, update their local map, and when they receive all the stop tasks from the other nodes, they send back to the Node 0 their local result (Figure 3).

The program, when possible, implements a non-blocking asynchronous communication to overlap communication and computation.

Since nodes communicate by sending complex objects (such as `element_t`), and MPI does not natively support structs, I had to define custom structures for both `element_t` and `curve` objects, carefully managing attribute offsets when defining the `MPI_Type`.

To know what type of object read from the receive operation the messages have been tested with the Probe operator, to know in advance the tag and the number of elements, because at different tags correspond different objects to read.

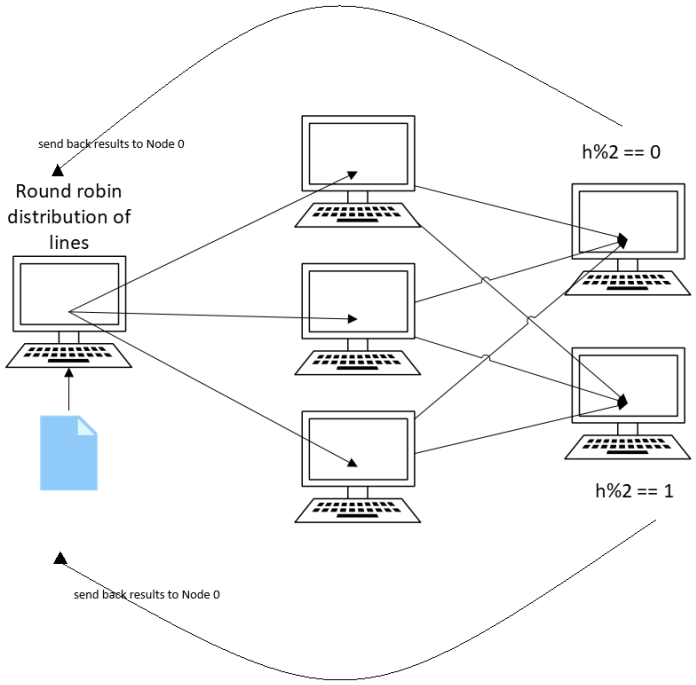

Figure 3: First final solution for the MPI version.

3.2 Solution 2

In the second solution (Figure 4), a master node (Node 0) first stores the input file and then distributes the lines in batches to all other nodes (including itself) using `MPI_Scatterv`. Each node, including Node 0, starts processing by parsing the lines and generating elements.

Each node stores its own elements—i.e., the elements for which the modulo operation returns the node's rank—and sends the remaining elements to the corresponding nodes using `MPI_Ialltoall`. This operation is non-blocking to allow computation and communication to overlap, in fact while the All-to-All exchange is still in progress, each node can process its local elements and count similarities. Once all nodes have received the elements from their peers, they continue the similarity counting.

Each cycle processes a batch of lines. At the end of the entire input file, Node 0 gathers the partial results from all nodes using `MPI_Gather`, sums them, and prints the final result.

Additionally, line parsing is performed using OpenMP's parallelism with a `parallel for` construct.

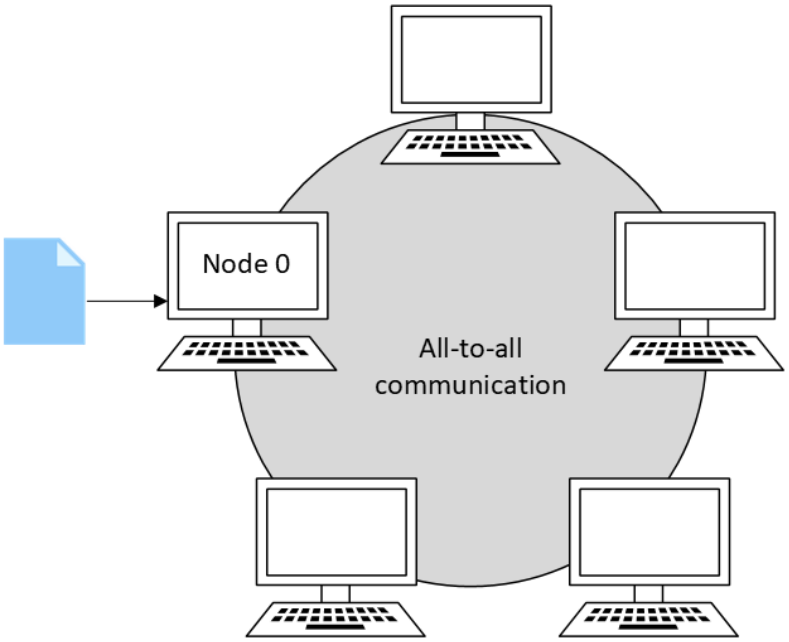

Figure 4: Second solution which uses an all to all communication paradigm

4 Results

All the tests have been executed on the cluster nodes with the following commands:

- sequential: `srun --nodes=1 ./sequential inputDataset`
- fastflow: `srun --nodes=1 ./fastflow inputDataset farm1 farm2`
- mpi: `srun --mpi=pmix --cpus-per-task=32 --nodes=n ./mpi inputDataset`

4.1 Fastflow results

For the FastFlow implementation, the best performing configuration is 7 parsers and 6 counters (Table 1), with only a minimal difference observed for the 5GB file between the (4,4) and (7,6) configurations. Testing with unbalanced farms confirms that a configuration where $Size(Farm1) > Size(Farm2)$ produces better performance compared to the opposite case, $Size(Farm2) > Size(Farm1)$.

It is important to note that the farm size does not directly correspond to the total number of nodes, as additional nodes are required for the emitter and collectors, leading to an overall increase of $+3$ nodes. Beyond the (7,6) configuration, no further performance improvements are observed, likely due to reaching the physical limit of 16 physical cores available (Figure 5) .

Additionally, the (2,2) configuration represents the even and odd version.

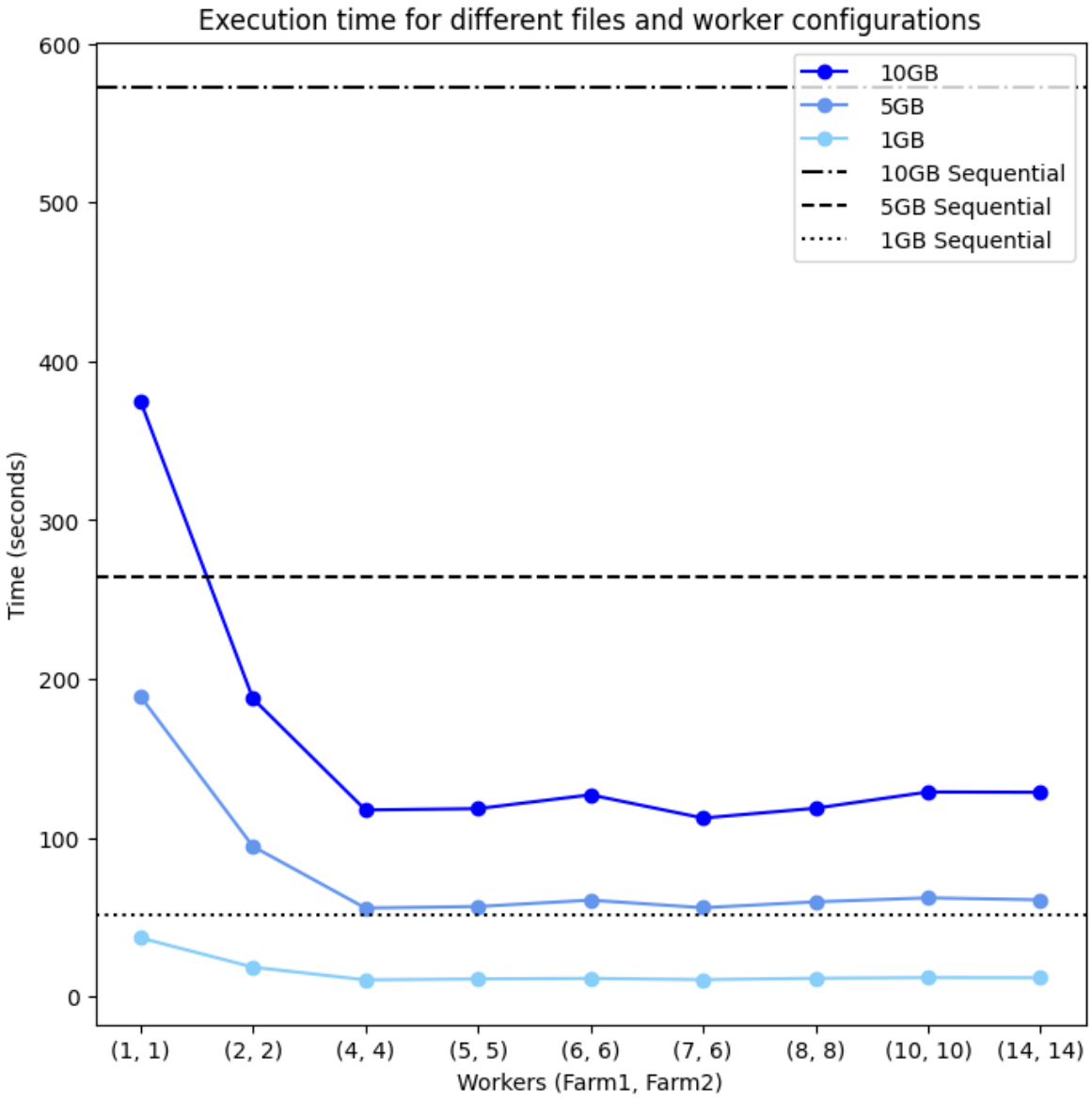

Figure 5: Execution time comparison for the fastflow program with batch=1000 versus the sequential one

File	Time (s)	Max Speedup	Efficiency	Workers configuration
1GB	10.76	4.82	43.8%	7,6
5GB	55.79	4.74	43.1%	4,4
10GB	112.45	5.09	46.3%	7,6

Table 1: Maximum speedup and configurations for different file sizes for fastflow solution.

4.2 MPI results

Tests for the first MPI version must have at least 4 nodes, since one is the reader, one is the parser and two are the even and odd nodes. As we can expect the second version

of the MPI algorithm performs better (See Figures 6 and 7, Tables 2 and 3), not just because it has a better structure but also because it uses internal parallelism.

Adapting the batch size to the input file can lead to better results, for example, the second version of the mpi algorithm obtains better results for the 10GB input file if we increase the batch size from 1000 to 5000.

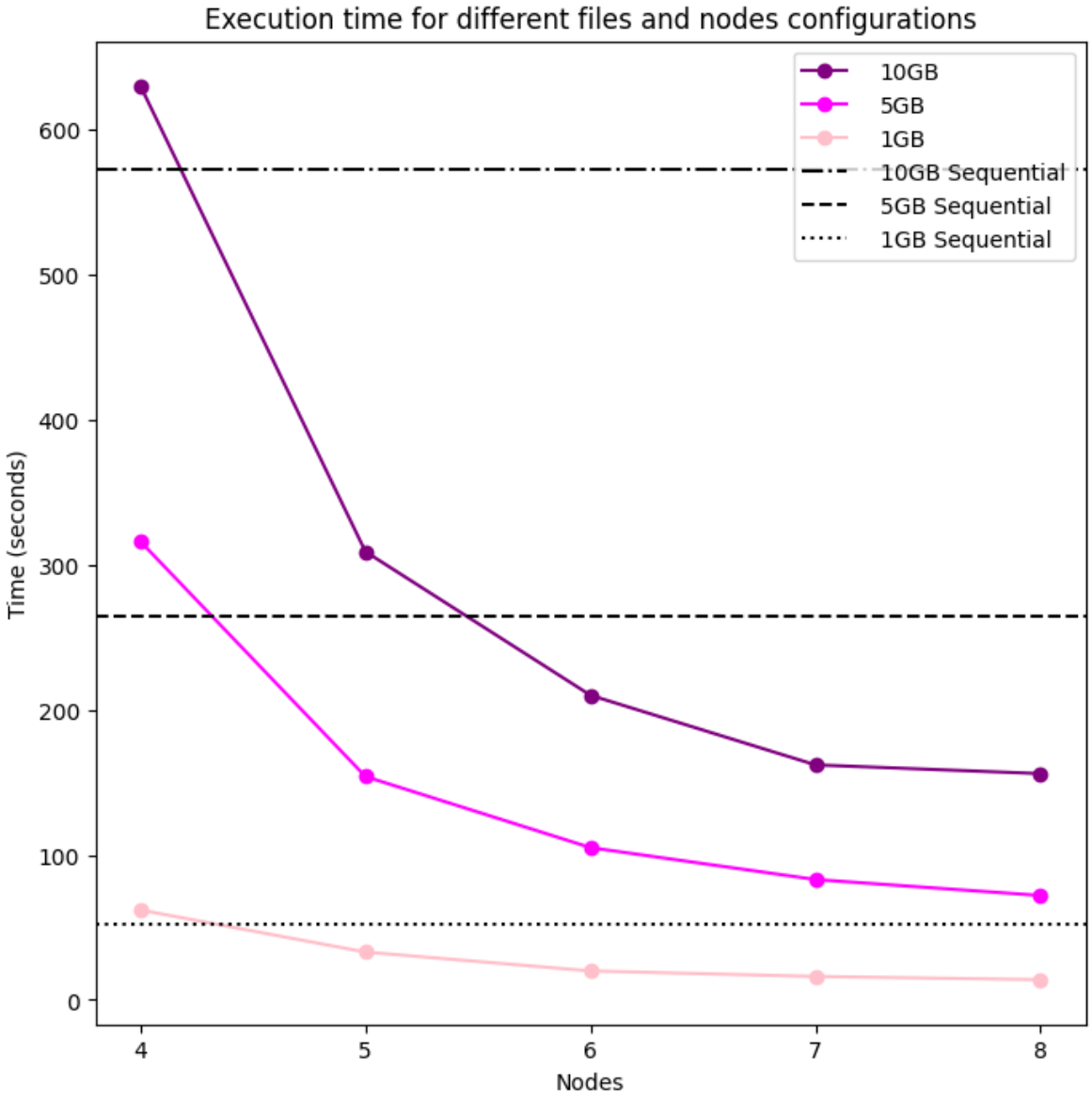

Figure 6: Execution time comparison between the first MPI solution and the sequential program

File	Time (s)	Max Speedup	Efficiency	Nodes
1GB	14.10	3.71	46.37%	8
5GB	72.35	3.67	45.8%	8
10GB	156.13	3.67	45.8%	8

Table 2: Maximum speedup and configurations for the first MPI solution.

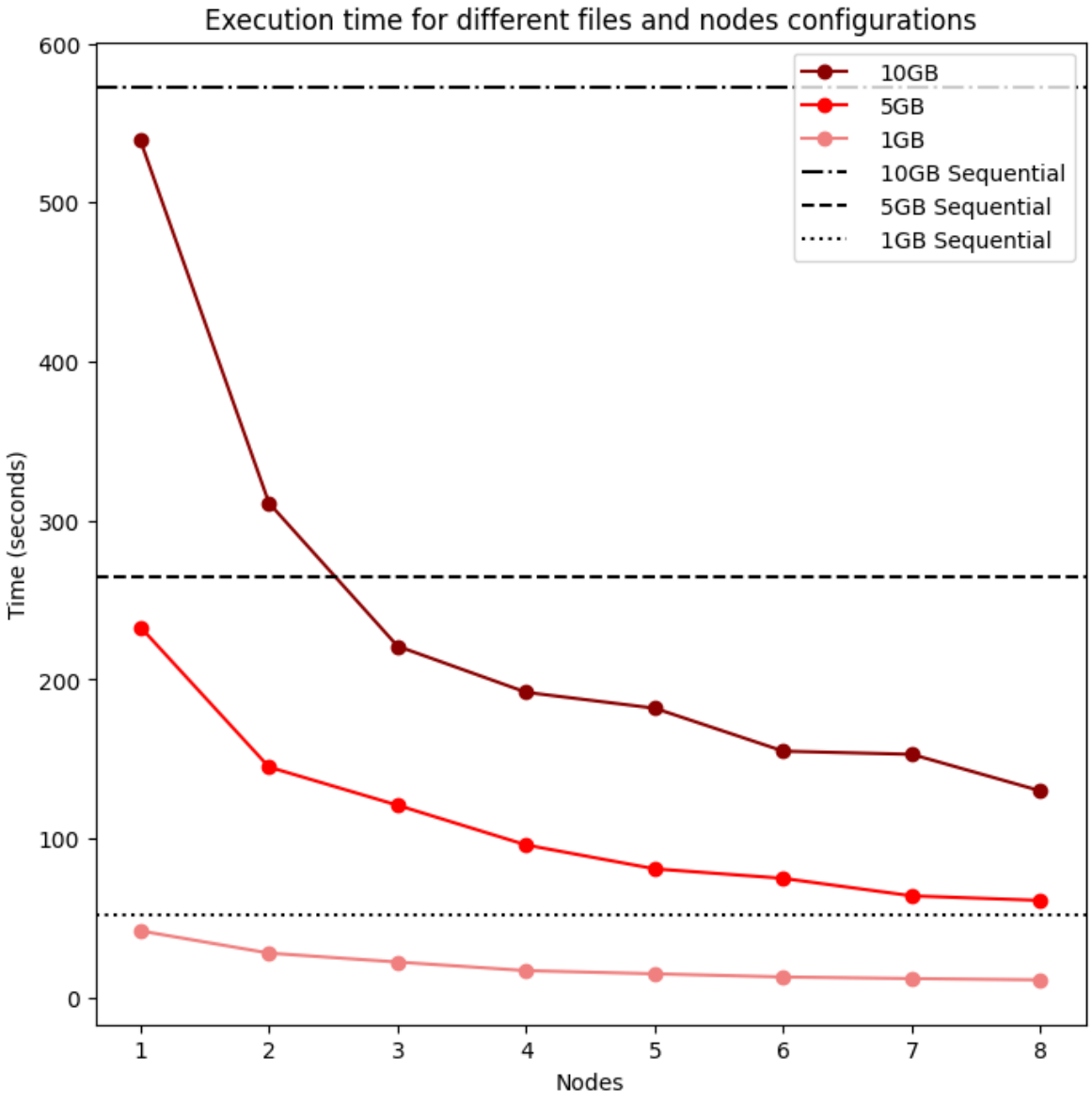

Figure 7: Execution time comparison between the second MPI solution and the sequential program

File	Time (s)	Max Speedup	Efficiency	Nodes
1GB	11.2	4.63	57.8%	8
5GB	61.15	4.32	54%	8
10GB	130.38	4.41	55.1%	8

Table 3: Maximum speedup and configurations for the second MPI solution

4.3 Comparison

The FastFlow implementation performs better than the MPI-based versions (Figure 8), likely due to the overhead introduced by MPI communications/synchronizations. The

objects being exchanged are complex, which can significantly impact communication efficiency. In particular, in the second MPI implementation, serialization and deserialization of objects are done before and after data transmission, increasing the overhead and potentially slowing down execution.

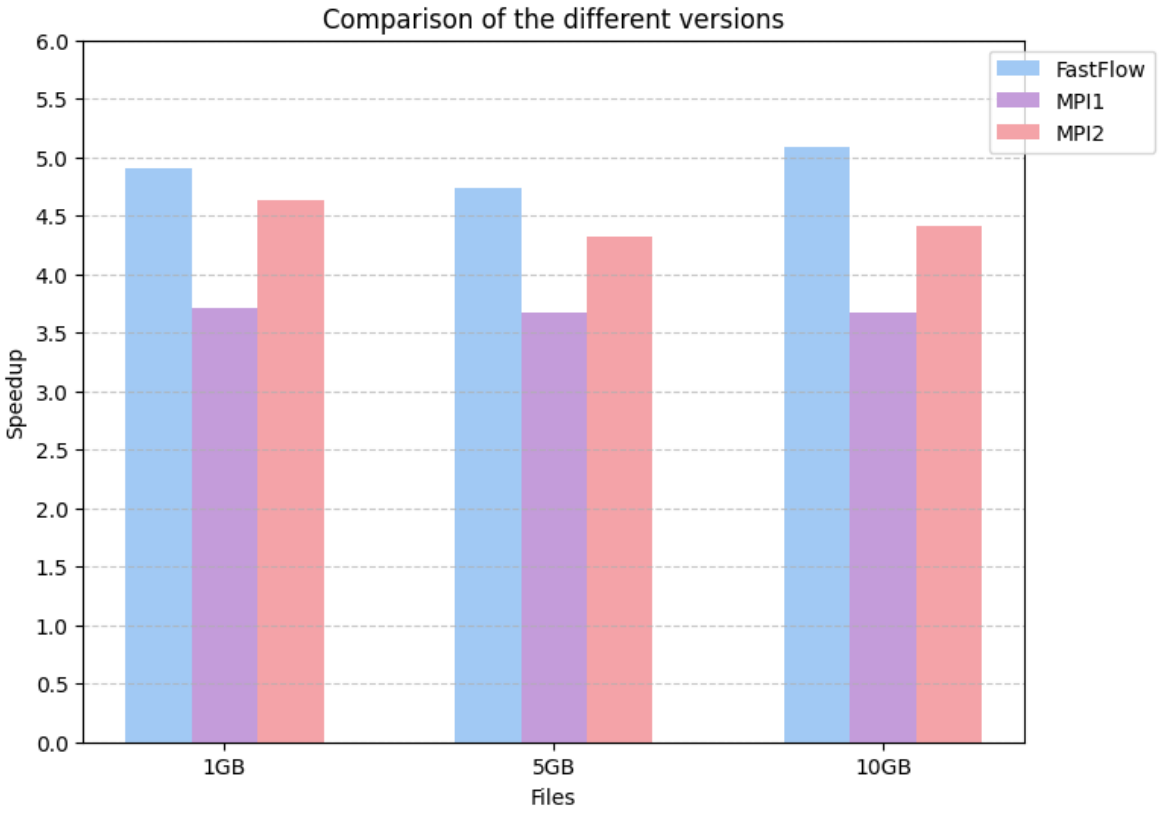

Figure 8: Comparison of the best speedup among all solutions for all the input files

5 How to run

Content of the project folder:

- dependencies: folder with math functions to calculate the LSH
- ff: fastflow folder
- sequential.cpp: sequential version of the algorithm
- fastflow.cpp: fast flow implementation
- mpi1.cpp: First MPI implementation
- mpi2.cpp: Second MPI implementation
- Makefile
- test_sequential.sh: script to execute complete sequential tests

- test_fastflow.sh: script to execute complete fastflow tests
- test_mpi1.sh: script to execute complete test of the first mpi version
- test_mpi2.sh: script to execute complete test of the second mpi version
- test_small.sh: fast tests on 1GB file for all the versions

Tests must be executed on the spmcluster machine since the input files are not copied in the project folder.

To execute small tests write the following command in the main folder:

- `make all`
- `sbatch ./test_small.sh`

Output file `small.out` will be generated in the output folder.