SPM*Assignment 4

MPI+FF

DAVIDE BORGHINI

June 2025

Contents

*In the end, the title was corrected

1 Introduction

This project focuses on the parallel implementation of the Mergesort algorithm using two different paradigms: **FastFlow** for shared-memory parallelism within a single node, and a hybrid approach combining **MPI** and FastFlow for multi-node execution. The aim is to study the problem that arise facing this problems, and possible solutions to them, and finally study the scalability and efficiency of the produced code on the `spmcluster`.

1.1 Cluster description

The experiments were conducted on the spmcluster, a high-performance computing cluster managed by the University of Pisa. The cluster consists of a single frontend/login node (`spmcluster.unipi.it`) and 8 compute nodes connected via a 10 Gigabit Ethernet network. Each node has two sockets. Each socket provides 8 physical cores and 16 virtual cores, so in total 16 physical cores and 32 virtual cores per node.

1.2 Problem Analysis

In Mergesort, the level of parallelism increases with each recursive call during the divide phase, reaching its maximum when the array is split into individual elements. During the merge phase, however, the available parallelism decreases at each level: with every merge step, the number of independent merge operations is halved. Once the number of subarrays to merge becomes fewer than the number of threads, each subsequent level can utilize only half the threads, eventually reaching a final merge that is inherently sequential and linear in the array size.

In practice, additional factors such as memory bandwidth limitations and, in the case of MPI, communication latency, can further constrain achievable performance. In the actual implementation, the divide phase is not explicitly parallelized. Instead, the array is partitioned into chunks of minimum size assigned to threads or nodes. This avoids excessive overhead from managing parallelism at fine granularity.

2 Implementations

All the implementations in C++ can be found on GitHub [1].

2.1 Sequential

The sequential implementation is straightforward and uses the `stable_sort` of the standard C++ library. This choice is done because this is likely one of the best implementation of the merge-sort reasonably simple to use. Although in some case it may not use merge-sort this is not really a problem since it will also be used in the parallel and distributed version.

2.2 Shared Memory

The sequential implementation was build incrementally and many choices were done by doing quick test to see which direction to give to the implementation, and on where to focus the optimization effort.

The first implementation is the straightforward naive implementation that uses a farm with an emitter and a 'wraparound'. This first version was a starting point that has 3 main problems: first the emitter was not working making the implementation highly inefficient (especially when using a small number of worker); the second problem

was the fact that, after sorting the various pieces in which the array was split, the merge was executed one level at the time, possibly reducing the parallelism; the final problem was the usage of memory that wasn't really efficient.

The first changes had the goal to solve the problem of limited parallelism, while reusing tasks consequentially reducing memory footprint of the program. Some test showed this version to reach the same performance of the original one or slightly worse while being more complex to implement.

The second variant aimed to maximize efficiency, making also the emitter work on tasks if all the workers had full queue. This implementation revealed faster and more efficient for low number of threads, results are showed in the experiment section 3. A last thing that was done to improve memory usage was the use of inplace merge that also slightly improved performances.

2.3 Distributed Memory

For the MPI implementation we first of all need to decide what we want to study, we can find ourself in multiple situation: the data are on a file that need to be sorted, the data are on a node and should remain there, or we can also decide that all the nodes should have the full array.

I decided to start with all the data in a single node, since this allowed a deeper study of the MPI programming paradigm and was more directly comparable with the shared memory version. The second decision was that of not sending the payload, this was done to minimize the communication overhead, and avoid sending data that only needed to be returned to the root node.

The code is designed to leverage the double buffering technique. However, in this context, it introduces a drawback: after independently sorting different portions of the array, the results must still be merged. This merging step is not required when using a simple `MPI_Scatter` operation [4], which directly distributes disjoint segments of the input array.

The implementation is organized into two main phases. In the first phase, each node sorts the data it receives and, when applicable, merges it with data obtained in previous steps. The second phase performs a global merge[1] in a logarithmic number of steps, where the number of active nodes is halved at each iteration until only the root node remains.

3 Experiments

For both the implementation the number of variables to analyse was incredibly high, so a full exploration of the parameter of the programs was impossible, and possibly not even interesting. For this reason the focus is on the main important features, such as the number of thread, the number of nodes, and the different implementation. The final goal of this experiments is to study for the two implementation: weak scaling, strong scaling and efficiency.

3.1 FastFlow

Three implementations have been tested: a naive version, and two optimized versions using a working emitter with queue sizes of 1 and 2. The program was executed on

[1] To keep the logic of the code simple in the merging phase, the number of node is supposed to be a power of 2.

problems of varying sizes, using different numbers of threads[2] and different task sizes[3]. The results are shown in Figure 2 1

Additional tests were performed using payloads of size 1 and 64. As expected, the payload size had no significant impact on performance, since in the implementation the payloads are represented as pointers to arrays and are not directly involved in the sorting process.

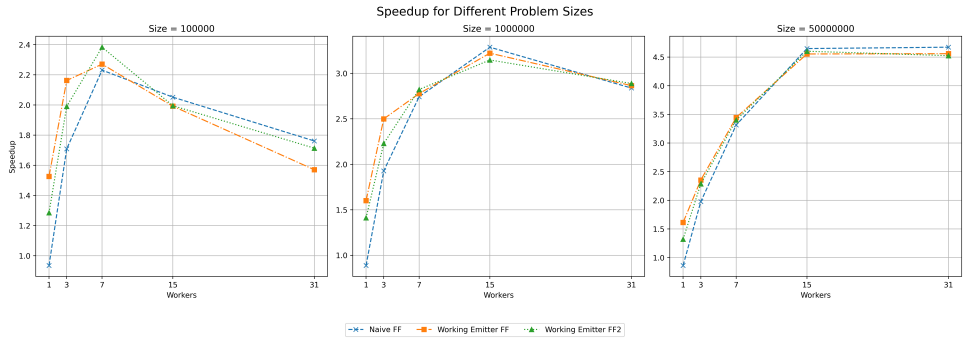


Figure 1: Graphs showing the speed-ups for the different implementations among three selected sizes, using the optimal task size.

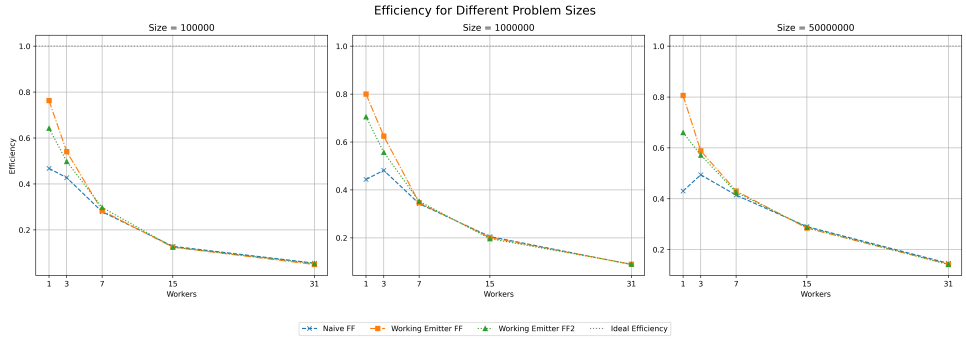


Figure 2: Graphs showing the efficiency for the different implementations among three selected sizes, using the optimal task size.

3.2 MPI + FastFlow

Several incrementally complex implementations using MPI were developed. The first version uses only MPI combined with a sequential sorting algorithm. The second introduces intra-node parallelism using FastFlow, and the final version extends this by employing double buffering to attempt overlapping computation and communication.

Figure 3 reports the execution times for all implementations, including a configuration where two MPI processes were run per node. Figure 4 presents the strong scaling results for the sequential MPI version and the best-performing parallel implementation.

Efficiency and speed-up metrics are not reported here, as none of the MPI implementations outperformed the purely sequential version except when all 8 nodes were used. For this reason, further analysis of speed-up and efficiency wasn't considered meaningful in this context.

[2]The number of threads used is the number of workers + 1

[3]The task size is the minimal unit of computation assigned to a worker for sorting.

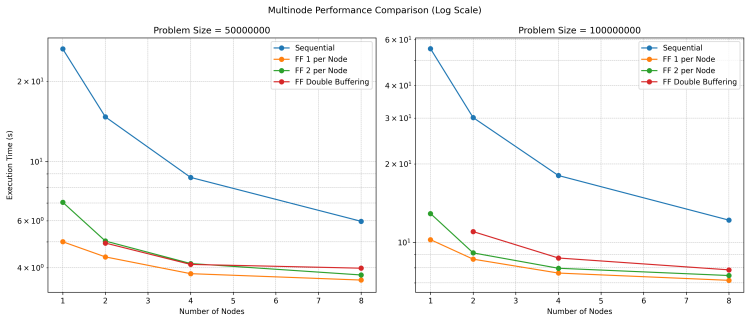


Figure 3: Graphs showing the time for the different MPI implementation an 2 different problem size.

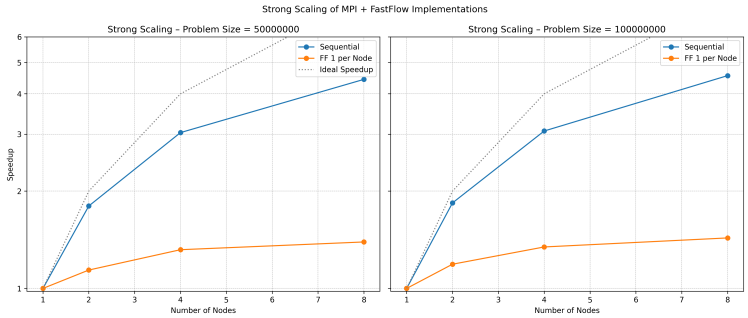


Figure 4: Graphs showing the scalability for the different implementations.

4 Analysis

4.1 FastFlow

From the plots, we observe that the speed-up improves significantly with increasing array size. This is likely because the overhead of creating the FastFlow farm becomes negligible compared to the actual computation time. However, depending on the problem size, adding more threads beyond a certain point becomes counterproductive. For example, in the first plot of Figure 1, the speed-up decreases for all implementations beyond 8 threads. In the second plot, performance starts to decline after 16 threads, and in the third plot, the speed-up plateaus beyond that point. This behaviour may be attributed to the limited benefit of hyper-threading for Mergesort and the use of busy-waiting in the worker threads[4].

The speed-up is far from linear, and several factors contribute to this. First, managing threads introduces overhead that is only amortized with sufficiently large inputs. Second, the sequential baseline (C++ `std::stable_sort`) is highly optimized and difficult to outperform. Finally, the algorithm's structure imposes inherent limits on parallelism: the final merge phase cannot be fully parallelized. At each level of the merge tree, the number of concurrent merge operations is halved, which reduces parallelism toward the end of execution. The overall speedups look similar to those obtained by Atanas Radenski with an OMP implementation [6].

To better understand the relative cost of sorting and merging, a quick test was performed locally using a 5-thread setup (4 workers and 1 emitter), with a task size of 1 million and an input of 50 million records. The command used was:

[4]This hypothesis would require further testing to confirm.

```
./s_work_em_ff.a -r 1 -s 50000000 -k 1000000 -t 4
```

The measured times were:

- Creating the farm: `8.06e-05` seconds
- Initial sorting (first chunk per worker): `3.043` seconds
- Total time spent in the sorting phase: `2.648` seconds
- Overall time for `sort_records`: `5.696` seconds

These results show that local sorting accounts for a large share of the total runtime, with merging still representing a significant cost. This clarifies that both phases play an important role in performance, and neither can be overlooked. It also highlights the importance of tuning task size and thread count to minimize idle time and maximize resource utilization.

Another important aspect of the analysis is the efficiency of the different implementations. From the plots in Figure 2, it is evident that there is no single implementation that consistently outperforms the others across all thread counts. In general, for a small number of threads, the most efficient approach is the working emitter with a queue size of 1. This is particularly noticeable when using only one worker: in the naive implementation, a dedicated thread merely instructs the worker, effectively wasting one core. In contrast, the working emitter actively contributes to the sorting work, resulting in better speed-up and significantly improved efficiency.

As the number of threads increases, the benefit of using a working emitter diminishes. At 32 threads, for example, its efficiency becomes slightly worse compared to the naive version. A possible explanation is that as the emitter checks all the queues and completes its own sorting tasks, other threads may spend time idle, waiting for work. Furthermore, when the number of threads exceeds the number of physical cores (i.e., in oversubscribed scenarios), the emitter may be descheduled while sorting, introducing delays and reducing overall performance.

4.2 MPI + FastFlow

Initial tests suggest that increasing the number of MPI processes per node does not lead to performance gains. This can be explained in two ways: the first is that the use of threads caused over subscription of the CPU cores[5]. However, a similar trend—though less pronounced—is observed even when each process spawns only 8 threads. This suggests that the performance degradation might be due to network limitations that prevent efficient concurrent data transfers.

Additionally, the use of double buffering did not result in performance improvements. Considerable effort was made to minimize resource usage: buffers are allocated once at program startup, sized appropriately to avoid reallocations, and for the largest buffer (in the process with rank 0), the original array is reused. One area that could have been optimized further is the usage of the FastFlow farm. Currently, it is destroyed and recreated at each communication step. While this may introduce some overhead, it is unlikely to be the main bottleneck—if it were, we would expect the single-node implementation to show no speed-up at all. The likely cause of the performance drop is the additional merge step required to combine the arrays received by the root process. The cost of this merge appears to cancel out the potential benefits of overlapping communication and computation using double buffering.

[5] In the plots shown here, each process spawns 16 threads.

Despite this failed attempt at improving runtime, we can observe from Figure 4 that the version that uses MPI scales quite well, and significantly better than the version that also leverages intra-node parallelism using FastFlow. This is likely to happen because the latter version is much faster, and therefore the overhead introduced by the communication is more dominant. A similar behaviour can be observed in [6].

5 Conclusions and Possible Further Optimizations

The FastFlow implementation achieved a good level of performance, and there appears to be limited room for further optimization. However, some areas remain worth investigating. For example, it could be interesting to evaluate the impact of using blocking queues, especially when oversubscribing CPU cores. Another unexplored aspect is thread pinning; different pinning strategies, as seen in a previous assignment, can have a notable effect on performance.

As for the MPI version, the performance curves for the two tested problem sizes are nearly identical. This is likely due to the sizes being too close to reveal meaningful differences. Future work could explore a wider range of input sizes or test the effect of completely removing the payload, transmitting only the keys to sort, in order to evaluate the communication overhead caused by sending the pointers.

Another useful direction would be to measure the execution time of each phase in the MPI implementation and analyse the time spent waiting during the double-buffering version. This could provide valuable insights into where improvements might be made.

Interestingly, almost the opposite behaviour was observed in [5], where the MPI version on four nodes outperformed an OpenMP version using four threads. The authors explain this by noting that communication overhead in MPI is less impactful than memory access latency in OpenMP when data no longer fits in cache. This suggest an interesting direction for future studies, although a different cluster architecture may result in different findings.

Lastly, the non-double-buffered MPI implementation could benefit from applying similar memory management strategies used in the double-buffered version. Moreover, the double-buffering strategy itself could be revised to perform the merging of the received chunks in parallel, as now it is computed by the main thread after the sorting of each chunk.

6 Appendix

6.1 Weak Scaling

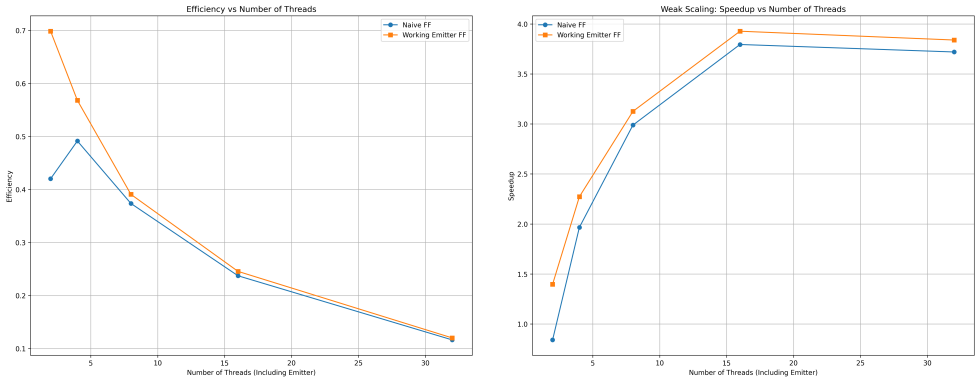
Since the cost of sorting is not linear with respect to the input size, the number of elements n in the array was computed by solving the equation $n \log n = p \cdot c$, where p is the number of threads and c is a constant determined using a base problem size and a reference thread count. This approach ensures that the computational load grows proportionally with the number of threads, accounting for the $O(n \log n)$ complexity of the sorting problem.

6.1.1 FastFlow

For what concerns FastFlow, from Figure 5 we can find similar trends to the ones observed in section 3.1: efficiency higher for the version with the working emitter, and

a speed-up[6] with more or less the same values. The only difference seem to be the fact that the naive implementation always looses to the version with the working emitter.

A reasonable explanation for the early loss of efficiency and the sublinear speed-up, as previously discussed in the analysis section, lies in the inherent limitations of the final merging phase. Since this step cannot be fully parallelized, it prevents complete utilization of all threads, particularly as the number of threads increases.



(a) (b)

Figure 5: Weak speed-up results for FastFlow (c=9965784). (a) Efficiency for both implementations. (b) Corresponding speedup curves.

6.1.2 MPI + FastFlow

In Figure 6, we observe a plot that closely resembles the one discussed in Section 3.2. This similarity can be explained—perhaps unsurprisingly at this point—by the cost of the merging phase.

To achieve linear speedup in a weak scaling setup, the sequential portion of the program should grow more slowly than the parallel portion. However, in this case, it effectively doubles[7] due to the cost of the final merge step, which remains largely sequential and grows with the overall problem size.

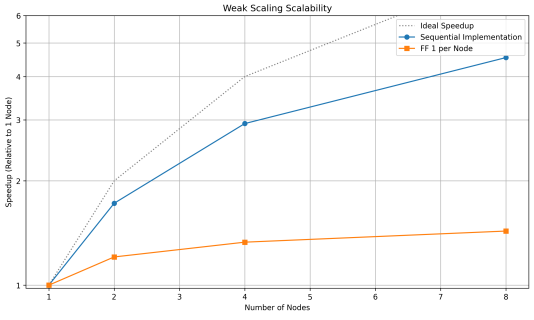


Figure 6: Weak scaling results for the MPI implementation. (c=332192809)

[6]Here speed-up is considered instead of scaling since it does not make sense to use a farm with only one thread that is the emitter.

[7]Not exactly doubles, since we are not strictly doubling the array size.

References

[1] GitHub repo containing the code: GitHub. (2025)

[2] Documentation for MPI_Finalize: DOC. (2025)

[3] Documentation for MPI_Receive: DOC. (2025)

[4] Documentation for MPI_Scatter: DOC. (2025)

[5] Alghamdi, T; Alaghband, G. High performance parallel sort for shared and distributed memory MIMD: arxiv. (2019)

[6] Radenski, A. Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs Champman University. (2011)

[7] Slides on MPI: MS Teams. (2025)