Assignment 4 report
SPM course
2024/25

Francesco Scarfato

June 5, 2025

1 Objective

The goal of this project is to implement a parallel merge sort by combining FastFlow's basic blocks with OpenMPI's distributed communication. Using both libraries, we aim to accelerate sorting across multi-core and multi-node environments.

2 Implementation

The codebase is organized into two source files. The first file focuses exclusively on a FastFlow-based merge sort, while the second one integrates MPI with FastFlow to distribute data among processes and apply FastFlow's intra-process parallelism.

2.1 FastFlow implementation

In the FastFlow-only version, we explore two farm-based strategies to parallelize sorting and merging.

- **Simple Farm Strategy**: begins with an Emitter that splits the input array into n sub-arrays. Each Worker sorts its assigned sub-array in parallel, and a single Collector gathers all sorted sub-arrays for a final, sequential merge.

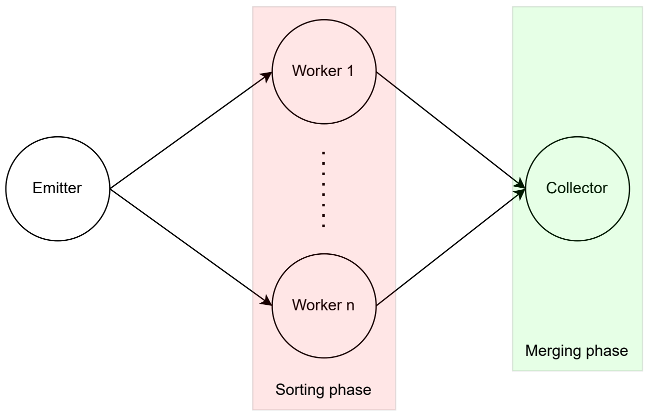

Figure 1: Simple Farm Strategy

- **Double Farm Strategy**: extends this idea by introducing a second farm stage for merging. After the initial farm's, the Collector receives the sub-arrays and it becomes the Emitter for a second farm. In this second stage, each Worker merges pairs of sorted chunks in parallel, effectively halving the number of sorted sequences. If more than two sequences remain after that step, the final Collector performs a sequential merge. A FastFlow pipeline wraps these two farms.

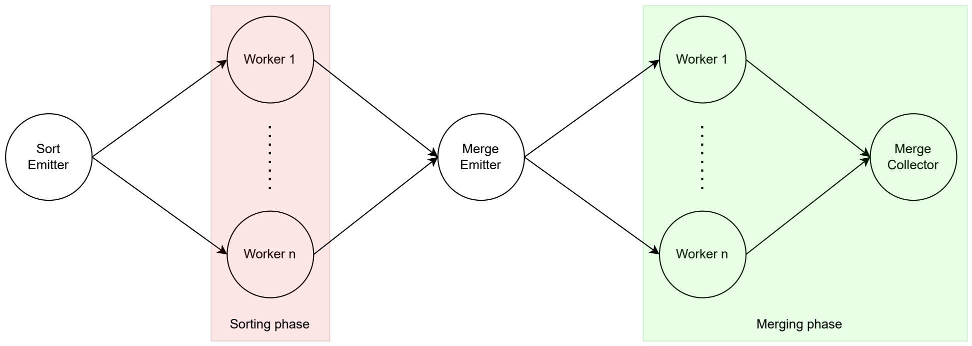

Figure 2: Double Farm Strategy wrapped in a pipeline

2.2 MPI with FastFlow Implementation

The hybrid version combines MPI for inter-process communication and FastFlow for intra-process parallelism. Initially, the data is distributed across MPI processes using `MPI_Scatterv`, ensuring a balanced load even when the total number of elements is not divisible uniformly among nodes.

Once each process receives its portion of the array, it performs local sorting using FastFlow. After sorting, two different strategies are employed to merge the results, depending on the number of processes:

- **Sequential Merge (Non-Power-of-Two Process Count):** If the number of MPI processes is not a power of two, all sorted sub-arrays are gathered at the root process using `MPI_Gatherv`. The final merge is then performed sequentially on the root node.

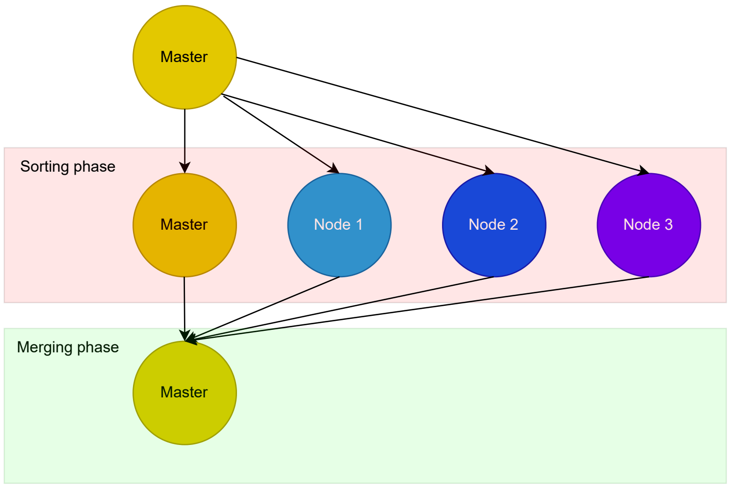

Figure 3: Merge with Gatherv

- **Tree-Based Merge (Power-of-Two Process Count):** If the number of processes is a power of two, a distributed merge is performed using a tree-based reduction strategy. The total number of merge levels is computed as $\log_2(n)$, where n is the number of processes. Processes are grouped into pairs at each level: one process sends its sorted sub-array, while the other receives it and merges it with its own. This continues recursively until the master process holds the fully merged array.

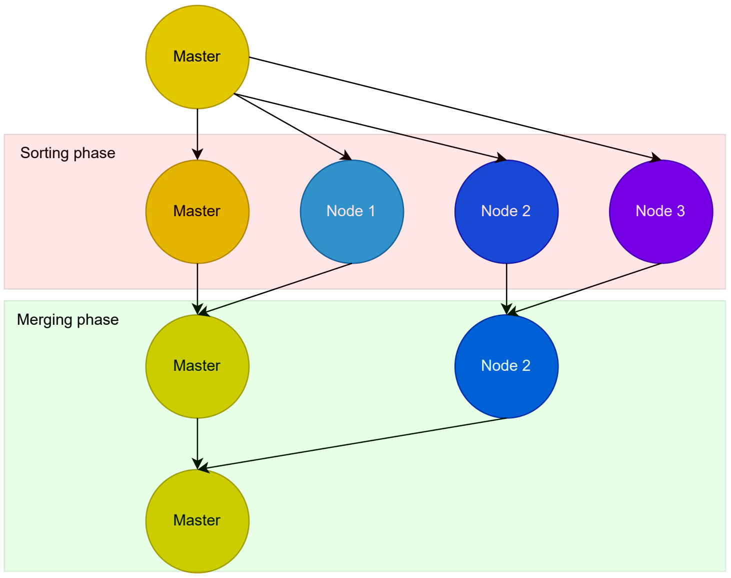

Figure 4: Tree-based merge

3 Performance analysis

In this section, we conducted studies on the two FastFlow-based strategies and on the MPI strategy. All tests were conducted on the internal computational nodes. Each experiment was repeated three times, and we report the arithmetic mean of those runs.

3.1 FastFlow-only

In this subsection, we compare the two FastFlow-based parallel mergesort strategies.
We vary the number of threads, the input size, and the payload size in order to understand scaling behavior. Note that, by design, payload size does not affect the measured execution times because the payload is stored in the master thread and never moved during mergesort, which operates only on record keys.
Both approaches show good scalability up to 8 or 16 threads, after which speedup and efficiency degrade noticeably. This decline is mainly due to synchronization overheads and the increasing cost of merging phases that become bottlenecks as the number of threads grows.
In the simple farm strategy, merging is centralized in the collector, which creates a merge bottleneck. The two-farm strategy attempts to alleviate this by parallelizing the first merging level, but since subsequent merging is still sequential in the second collector, this only partially mitigates the issue.
One major optimization adopted in the implementations is the management of payload data: instead of moving entire payloads during sorting and merging, the payload is stored once in a map on the master thread. The merge-sort operates only on keys, avoiding expensive data movement and reducing memory access. This design ensures that payload size does not significantly impact execution times, as confirmed by our tests.
Despite these optimizations, overhead from thread management, task scheduling, and memory access patterns limits scaling at higher thread counts, especially with 32 threads where contention and synchronization costs become dominant.
Below, we present tables and plots for three problems sizes (10^6, 10^7, and 10^8), comparing execution time, speedup, and efficiency for both strategies.

3.1.1 Size 10^6

Threads	Seq. Time (s)	Simple Farm			Double Farm		
		Par. Time (s)	Speedup	Efficiency (%)	Par. Time (s)	Speedup	Efficiency (%)
1	0.0873	—	—	—	—	—	—
2	—	0.0577	1.51	75.4	0.0582	1.50	75.0
4	—	0.0387	2.26	56.5	0.0410	2.13	53.3
8	—	0.0291	3.00	37.5	0.0300	2.91	36.4
16	—	0.0310	2.82	17.6	0.0265	3.29	20.6
32	—	0.0565	1.54	4.8	0.0534	1.63	5.1

Table 1: Comparison of the two strategies for size 10^6

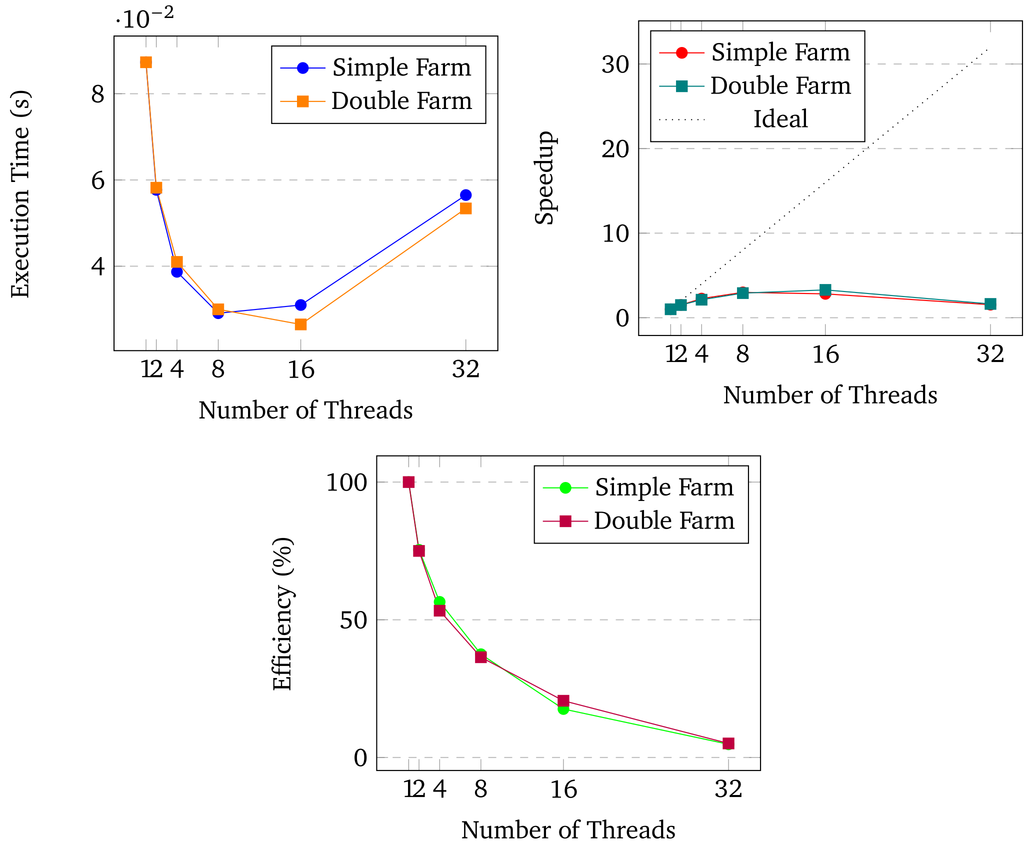

Figure 5: Performance metrics (Execution Time, Speedup, Efficiency) for size 10^6, comparing the two strategies

3.1.2 Size 10^7

Threads	Seq. Time (s)	Simple Farm			Double Farm		
		Par. Time (s)	Speedup	Efficiency (%)	Par. Time (s)	Speedup	Efficiency (%)
1	1.0336	—	—	—	—	—	—
2	—	0.6617	1.56	77.8	0.6671	1.55	77.0
4	—	0.4393	2.35	58.8	0.4254	2.43	61.0
8	—	0.3603	2.87	35.8	0.3358	3.08	38.5
16	—	0.3826	2.70	16.9	0.3347	3.09	19.3
32	—	0.5542	1.86	5.8	0.3997	2.59	8.1

Table 2: Comparison of the two strategies for size 10^7

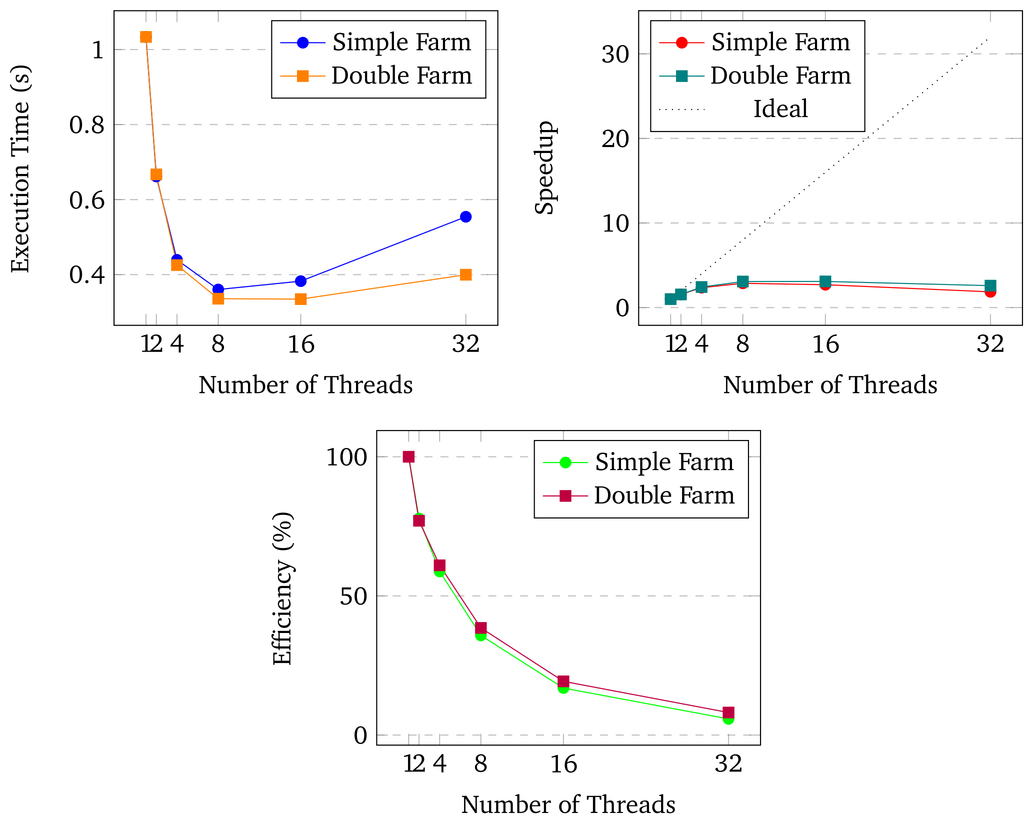

Figure 6: Performance metrics (Execution Time, Speedup, Efficiency) for size 10^7, comparing the two strategies

3.1.3 Size 10^8

Threads	Seq. Time (s)	Simple Farm			Double Farm		
		Par. Time (s)	Speedup	Efficiency (%)	Par. Time (s)	Speedup	Efficiency (%)
1	11.8979	—	—	—	—	—	—
2	—	7.5173	1.58	78.9	7.2240	1.65	82.5
4	—	4.8036	2.48	62.0	4.6273	2.57	64.3
8	—	4.0589	2.93	36.6	3.5554	3.35	41.9
16	—	4.5503	2.61	16.3	3.4712	3.43	21.4
32	—	6.5738	1.81	5.7	4.6460	2.56	8.0

Table 3: Comparison of the two strategies for size 10^8

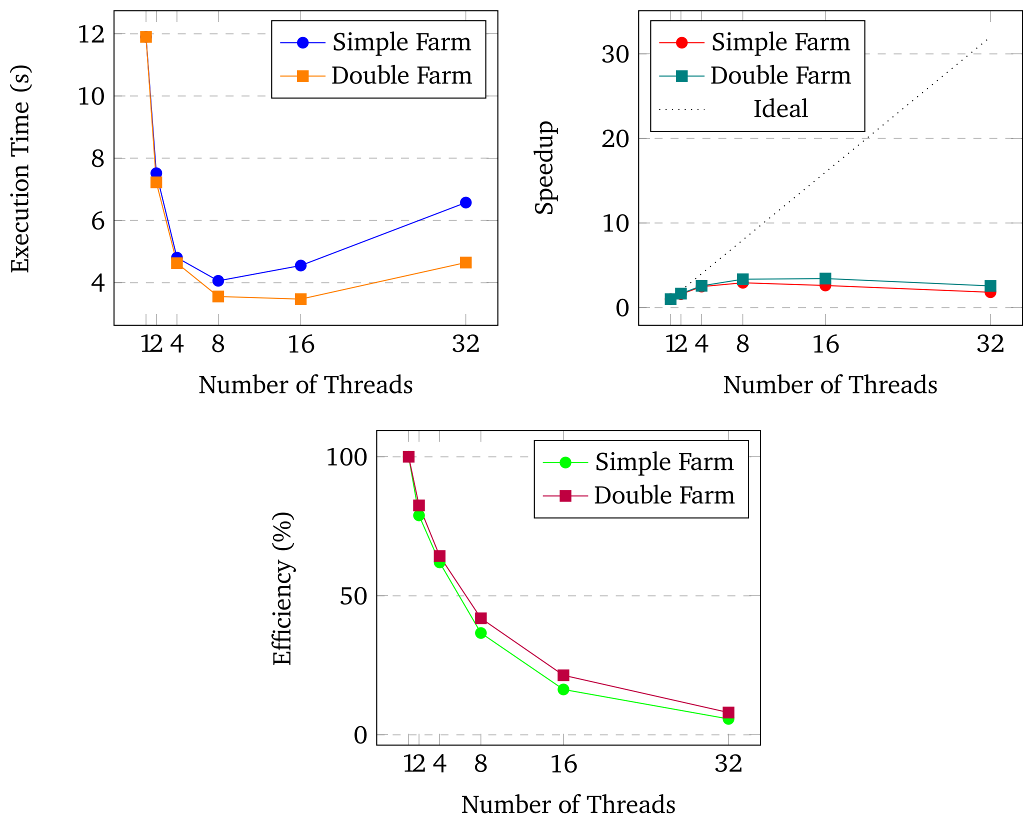

Figure 7: Performance metrics (Execution Time, Speedup, Efficiency) for size 10^8, comparing the two strategies

3.2 MPI with FastFlow

In parallel computing, we distinguish between *strong scaling* (fixed total problem size) and *weak scaling* (problem size grows proportionally with the number of processors). Ideally, with p processors we would like to finish p times more work in the same time (linear scalability: $S_p = p$). In practice, the communication costs and the sequential part limit this ideal behavior.

3.2.1 Strong Scaling (Fixed Global Size)

For strong scaling, we hold the global input size S constant and increase the total number of MPI processes $\times$ FastFlow threads.
Across all three sizes and all payloads, adding more cores (nodes $\times$ threads) *never* yields $S_p > 1$ for $p > 1$. Increasing the number of nodes and threads, increase slightly the speedup but the communication costs and the sequential part dominate.

Nodes	Threads	Cores	Payload = 1		Payload = 24		Payload = 256	
			Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1	1	1	0.1037	1.00	0.1040	1.00	0.1036	1.00
2	2	4	0.2411	0.43	0.5889	0.18	3.3167	0.03
4	2	8	0.2547	0.41	0.6224	0.17	3.3035	0.03
8	2	16	0.2828	0.37	0.6712	0.16	3.3399	0.03
2	4	8	0.2322	0.45	0.6133	0.17	3.3011	0.03
4	4	16	0.2509	0.41	0.5982	0.17	3.2867	0.03
8	4	32	0.2897	0.36	0.6386	0.16	3.3972	0.03
2	8	16	0.2286	0.45	0.5983	0.17	3.2988	0.03
4	8	32	0.2498	0.42	0.5963	0.17	3.3069	0.03
8	8	64	0.2917	0.36	0.6392	0.16	3.4265	0.03
2	16	32	0.2308	0.45	0.5796	0.18	3.2782	0.03
4	16	64	0.2487	0.42	0.6121	0.17	3.3010	0.03
8	16	128	0.2814	0.37	0.6411	0.16	3.3385	0.03
2	32	64	0.2514	0.41	0.6109	0.17	3.3295	0.03
4	32	128	0.2718	0.38	0.6172	0.17	3.3080	0.03
8	32	256	0.3125	0.33	0.6640	0.16	3.3511	0.03

Table 4: Strong-scaling results (time and speedup) for Size = 10^6.

Nodes	Threads	Cores	Payload = 1		Payload = 24		Payload = 256	
			Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1	1	1	1.2136	1.00	1.1991	1.00	1.2067	1.00
2	2	4	2.3142	0.52	5.9584	0.20	32.9449	0.04
4	2	8	2.3003	0.53	5.9659	0.20	32.9412	0.04
8	2	16	2.3010	0.53	5.9212	0.20	32.8973	0.04
2	4	8	2.2298	0.54	5.8383	0.21	32.8772	0.04
4	4	16	2.2532	0.54	5.8797	0.20	32.9079	0.04
8	4	32	2.2765	0.53	5.8663	0.20	33.1914	0.04
2	8	16	2.1653	0.56	5.7927	0.21	33.2022	0.04
4	8	32	2.2431	0.54	5.8696	0.20	32.9928	0.04
8	8	64	2.2518	0.54	5.8758	0.20	33.4080	0.04
2	16	32	2.2077	0.55	5.8721	0.20	32.8748	0.04
4	16	64	2.2424	0.54	5.8741	0.20	32.8792	0.04
8	16	128	2.2578	0.54	5.8994	0.20	32.9053	0.04
2	32	64	2.3244	0.52	5.9296	0.20	32.9530	0.04
4	32	128	2.2544	0.54	5.9175	0.20	32.9531	0.04
8	32	256	2.2968	0.53	5.9155	0.20	32.9897	0.04

Table 5: Strong-scaling results (time and speedup) for Size = 10^7.

Nodes	Threads	Cores	Payload = 1		Payload = 24		Payload = 256	
			Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1	1	1	14.0980	1.00	13.7097	1.00	14.0324	1.00
2	2	4	23.7534	0.59	59.9053	0.23	330.9130	0.04
4	2	8	23.3100	0.61	59.0058	0.23	328.6660	0.04
8	2	16	22.9124	0.62	58.8610	0.23	327.9620	0.04
2	4	8	22.5193	0.63	58.7167	0.23	330.9250	0.04
4	4	16	22.6500	0.62	58.7668	0.23	328.0970	0.04
8	4	32	22.7375	0.62	58.5906	0.23	329.8020	0.04
2	8	16	22.1562	0.64	58.1676	0.24	327.5420	0.04
4	8	32	22.5925	0.62	58.5208	0.23	328.7790	0.04
8	8	64	22.5310	0.63	60.2510	0.23	328.6090	0.04
2	16	32	22.3686	0.63	58.6308	0.23	329.0340	0.04
4	16	64	22.6026	0.62	60.6769	0.23	329.0430	0.04
8	16	128	22.5901	0.62	58.5028	0.23	329.2490	0.04
2	32	64	23.9043	0.59	59.9900	0.23	329.5770	0.04
4	32	128	23.1967	0.61	59.3219	0.23	329.6110	0.04
8	32	256	22.9507	0.61	58.8645	0.23	328.9900	0.04

Table 6: Strong-scaling results (time and speedup) for Size = 10^8.

3.2.2 Weak Scaling (Work per Core Fixed)

According to the definition of weak scaling, the problem size increases proportionally with the number of processing elements, and the ideal case assumes that the execution time remains constant as we scale out.
The observed execution times increase significantly with the number of nodes: doubling the number of nodes roughly doubles the execution time, indicating suboptimal weak scalability. This trend is also reflected in the speedup, which degrades rapidly as more nodes are added. The primary causes of this behavior are the communication and synchronization overheads. As the number of processing elements grows, these costs covers the benefits of parallelism, leading to poor scaling.

Nodes	Threads	Total Cores	Problem Size	Time (s)	Speedup
1	1	1	10M	1.2136	1.00
2	8	16	20M	4.3759	0.28
4	8	32	40M	9.0600	0.13
8	8	64	80M	18.5103	0.07
2	16	32	20M	4.4539	0.27
4	16	64	40M	9.1544	0.13
8	16	128	80M	19.0046	0.06
2	32	64	20M	4.6036	0.26
4	32	128	40M	9.3793	0.13
8	32	256	80M	18.7817	0.06

Table 7: Weak scaling results

Final analysis As the global size grows from 10^6 to 10^8, the fraction of time spent in purely local FastFlow sorting increases faster than the MPI and merge overhead.
Larger sizes reduce the relative impact of MPI latency and the sequential merge, so both strong and weak efficiency improve with increasing S.

4 Further Improvements

While the current implementation demonstrates the effectiveness of parallel merge sort using FastFlow, in MPI the communication overhead and the sequential part dominates over the benefits of the parallelism. Several enhancements can be considered to improve performance and scalability.
In the FastFlow parallel implementation, the merging phase could be extended beyond a single level. Currently, the double farm performs only one level of parallel merging before falling back to a sequential collector. A more scalable approach would be to implement a recursive, tree-based merge pattern across multiple levels, reducing the final sequential workload. Additionally, introducing feedback channels could enable more dynamic communication. The workers can send the merged sub-arrays back, through the feedback channels, to the `MergeEmitter` that re-directs to appropriate workers. This could improve parallelism and allow for a more flexible and adaptive merging process.
In the current MPI tree-based merge, nodes that send their sorted sub-arrays are no longer involved in the merge process. An alternative approach would be to allow every node to participate in computing the final sorted array. Although this increases the number of communications, the overall time complexity remains the same since the merge operations can still be performed in parallel at each level of the tree.