Assignment 4

Parallel MergeSort with FastFlow and MPI for Shared and Distributed Memory

Giuseppe Gabriele Russo 583744

June 20, 2025

Abstract

We present three MergeSort variants for hybrid shared- and distributed-memory systems: a sequential baseline, a tree-based MPI + FastFlow version, and a fully distributed implementation built on Parallel Sorting by Regular Sampling (PSRS). Each algorithm combines intra-node parallelism provided by FastFlow with inter-node coordination via MPI to sort large arrays of fixed-size records. We describe their data decomposition, communication, and merge strategies, then analyse their behaviour on a multi-node cluster. The study shows that removing centralised merges and balancing I/O through PSRS markedly improves scalability, whereas tree-based merging is constrained by memory pressure and shared-filesystem contention. We conclude with a discussion of trade-offs and avenues for future optimisation, such as overlapping communication with computation and adaptive partitioning.

1 Introduction

This report presents the design and evaluation of a hybrid parallel MergeSort algorithm developed for Assignment 4 of the SPM course (a.a. 2024/2025). The goal is to sort large arrays of fixed-size records efficiently using both shared-memory and distributed-memory parallelism, leveraging FastFlow for intra-node computation and MPI for inter-node communication.

Three progressively optimized versions of the algorithm were developed:

- **Plain version (`plain_sort`)**: a sequential baseline implementation that uses `std::sort` to sort an array of records entirely on a single node. It is used as a correctness reference and performance baseline.
- **Tree-based merge version (`par_merge`)**: a hybrid version where only the keys are extracted and distributed across nodes. Each MPI rank sorts its local keys using FastFlow with a sequential local merge. A global tree-based merge is then performed across MPI ranks using point-to-point communications.
- **PSRS version with parallel merge (`par_merge_psrs`)**: the final and most scalable version, in which FastFlow is used to sort and merge keys in parallel within each node. Then, a distributed merge is performed using the Parallel Sorting by Regular Sampling (PSRS) algorithm, resulting in globally sorted data split across ranks.

All implementations work on arrays of records with a sortable `key` field and a fixed-size payload. To ensure reproducibility, all datasets are pre-generated and loaded from binary files.

Sorting is performed only on the key arrays, and the full `Record` structures are reordered at the end according to the final sorted order. Finally, the sorted array is written to a new binary file to allow validation and subsequent processing.

The report details the architecture, communication patterns, and trade-offs of each version, and concludes with a performance evaluation comparing speedup, efficiency, and scalability on the `spmcluster` platform.

2 Data Structure and Problem Specification

The sorting problem tackled in this assignment involves large arrays of structured data, where each element is a `Record` composed of a sortable key and a payload. The payload is not involved in the sorting process but must be preserved. The data structure used throughout all implementations is the following:

```
struct Record {
    unsigned long key;
    char rpayload[RPAYLOAD];
};
```

The key is a 64-bit unsigned integer, while the payload is a fixed-size buffer of bytes whose size is determined at compile time. Since the payload is opaque and unused in comparisons, sorting focuses exclusively on the `key` field.

To ensure consistency across different experiments and avoid variability due to random generation, all datasets are pre-generated and stored as binary files. Each execution loads the data from these files using the `load_data()` function, sorts the array, and then writes the result to a new binary output file. This design not only improves performance by reducing runtime initialization overhead but also simplifies correctness verification by allowing direct comparison of output files.

Sequential Baseline (`plain_sort`)

In the baseline version, the full array of `Record`s is loaded into memory on a single node and sorted sequentially using `std::sort` with a comparator on the key. Once sorted, the result is written back to disk. This version serves as both a reference for correctness and a baseline for evaluating speedup.

Hybrid Tree-Based Version (`par_merge`)

In this version, the sorting is split across multiple MPI ranks. Rank 0 is responsible for loading the entire dataset, from which only the keys are extracted and distributed to all ranks via `MPI_Scatterv`. Each rank sorts its local portion in parallel using FastFlow. Once local sorts are completed, the ranks perform a global merge using a tree-based merging scheme. The result is gathered at rank 0, where the original `Record`s are reordered according to the sorted key array using a map from key to position. The final sorted dataset is then written to a binary output file.

PSRS-Based Version (`par_merge_psrs`)

This final and most scalable version adopts a fully distributed strategy from the beginning. Each rank is assigned a chunk of the global dataset and independently loads its corresponding records from the binary input file. After a parallel local sort using FastFlow, the ranks engage in the PSRS algorithm to redistribute the data such that each process ends up with a globally sorted segment. The final result is partitioned across ranks and already in sorted order globally. Each rank writes its sorted segment to disk, and the full sorted dataset can be reconstructed simply by concatenating the per-rank outputs.

Command-Line Interface

All implementations accept the same set of command-line arguments to ensure ease of testing and consistency:

- `-s N` Total number of records to be sorted (e.g., `10M`, `100M`).
- `-r R` Payload size in bytes for each record (e.g., `8`, `64`, `256`).
- `-t T` Number of worker threads to use for FastFlow within each node.

The input file is automatically inferred based on size and payload parameters, and the sorted result is written to a new file with an appropriate suffix to indicate completion.

Command-Line Interface

All versions accept the following command-line arguments:

- `-s N` : total number of records (e.g., `10M`, `100M`).
- `-r R` : payload size per record (e.g., 8, 64, 256 bytes).
- `-t T` : number of FastFlow threads per node.

3 Sequential Implementation

The `plain_sort` version represents the baseline sequential implementation and serves as a reference for both correctness and performance benchmarking. It operates entirely on a single compute node, where the full array of `Record` structures is loaded from a binary file, sorted, and written back to disk.

Sorting is performed using the standard C++ library's `std::sort`, leveraging a custom comparator that operates solely on the `key` field. The payload is treated as opaque data and preserved throughout the process, but not considered during comparisons:

```
std::sort(v, v + n, [](const Record &a, const Record &b) {
    return a.key < b.key;
});
```

The sort is done in-place, without duplicating the array. While a temporary buffer may be allocated by the runtime or for validation, no explicit extra memory is required for the merge itself. This keeps memory overhead low and benefits from the optimized internal logic of the STL implementation.

Once sorting is complete, the result is validated against the original key ordering to ensure correctness, and the sorted records are then saved to a new binary output file. Timing measurements are collected around the sorting phase using high-resolution clocks to establish an accurate baseline for later comparisons.

This version is used throughout the evaluation for two main purposes:

- **Correctness**: its output provides a ground truth against which all parallel versions are compared.
- **Performance**: its runtime serves as the baseline in speedup and efficiency metrics.

Despite its simplicity and strong performance for moderate input sizes, this version becomes a clear bottleneck as data volumes grow. This justifies the need for parallel solutions that can scale across cores and nodes to handle larger datasets more efficiently.

4 Single-Node Parallel Implementation (FastFlow)

Both hybrid versions use FastFlow to perform intra-node parallel sorting. The implementation is structured around a `farm` pattern, where the dataset is divided into `T` disjoint chunks (with `T` the number of threads), and each chunk is assigned to a worker thread. The emitter also acts as a worker to maximize thread utilization.

The sorting algorithm applied by each worker is `std::sort`, and the farm behavior diverges depending on the specific version.

Version `par_merge`: Sequential Merge Without Feedback

In the `par_merge` version:

- Only the `key` field is used during sorting. The full `Record` structures are ignored during this phase.
- The emitter splits the key array and dispatches each segment to a worker.
- No collector or feedback channel is used: workers sort their segments in-place.
- After all workers finish, the emitter performs a sequential merge of the sorted segments.

This version is conceptually simple and works well for moderate thread counts. However, the final merge introduces a sequential bottleneck that limits scalability.

Version `par_merge_psrs`: Parallel Merge With Feedback

The `par_merge_psrs` version improves upon this design by performing both sorting and merging in parallel on full `Record` structures. Key characteristics include:

- The farm uses a **feedback channel**, which allows workers to return intermediate merge results to the emitter.
- The sorting phase still uses `std::sort`, but directly on `Record[]` arrays.
- After sorting, the emitter coordinates multiple parallel merge rounds: workers are paired to merge blocks recursively.

This design allows the merge phase to be fully parallelized, removing the sequential bottleneck and improving intra-node scalability.

Summary of Differences

- **Sorted Data**: `par_merge` works on keys only; `par_merge_psrs` sorts full records.
- **Emitter Role**: In `par_merge`, the emitter merges all data sequentially. In `par_merge_psrs`, it orchestrates merge rounds.
- **Feedback Channel**: Not used in `par_merge`; essential in `par_merge_psrs`.
- **Scalability**: The feedback-based version scales better as merge operations are distributed across threads.

5 Hybrid MPI + FastFlow Implementation

The hybrid versions of the sorting algorithm combine intra-node shared-memory parallelism using FastFlow with inter-node distributed-memory communication using MPI. Two hybrid implementations were developed with distinct communication and data distribution strategies.

5.1 Version `par_merge`: Tree-Based MPI Merge

In the first version, the data distribution and coordination are centralized. The execution proceeds as follows:

1. **Dataset loading**: rank 0 loads the entire array of `Record` from file using `load_data()`.
2. **Key extraction**: rank 0 extracts all `key` fields from the records into an array of `unsigned long`. A map `key → position` is built to reconstruct the original records after sorting.
3. **Distribution**: the key array is partitioned and scattered to all ranks using `MPI_Scatterv`. Each rank receives a chunk of the global key array.
4. **Local sorting**: each rank uses FastFlow to sort its chunk of keys in parallel. The merge at this stage is sequential and handled by the emitter.
5. **Global merging**: a tree-based merging algorithm (`tree_merge`) is used. At each round:
    - Odd-numbered ranks send their sorted array to the previous rank.
    - Even-numbered ranks receive and merge, continuing to the next level.

   This process continues until rank 0 receives and merges all key arrays.
6. **Final reconstruction**: rank 0 reconstructs the final sorted `Record` array by using the `key → position` map and the original records. The sorted array is then saved to a binary output file for downstream usage or validation.

While this approach is conceptually simple, it suffers from scalability limitations:

- Communication is asymmetric and centralized at rank 0.
- Merge is sequential at the root, becoming a bottleneck.
- Memory pressure is high on rank 0.

5.2 Version `par_merge_psrs`: PSRS-Based Distributed Merge

To overcome the limitations of the previous version, a more scalable solution was implemented based on the Parallel Sorting by Regular Sampling (PSRS) algorithm. This version distributes work and memory evenly across all ranks.

Execution proceeds as follows:

1. **Initial loading**: each rank independently loads its own chunk of `Record` structures from file, using a consistent split based on rank and total dataset size. No node holds the full array.
2. **Local sorting**: each rank sorts its local chunk of `Records` using FastFlow, with a parallel merge strategy supported by a feedback-enabled farm.
3. **Sampling and splitter selection**:

- Each rank selects regular samples from its local sorted data.
- Samples are gathered at rank 0 via `MPI_Gather`, sorted, and splitters are selected.
- Splitters are broadcasted back to all ranks using `MPI_Bcast`.

4. **Data redistribution**:
   - Each rank partitions its data into `P` buckets based on the splitters.
   - All-to-all exchange is performed via `MPI_Alltoallv`, so that each rank receives all the records belonging to its global partition range.
5. **Final sort**: each rank concatenates the received buckets and performs a standard `std::sort` on the resulting array of records. Since all records fall within a unique global range, this guarantees a globally sorted partition.
6. **Result structure**: the final global sorted array is distributed: rank 0 contains the smallest values, rank 1 the next set, and so on. Concatenating all local arrays yields the full sorted result. Each rank writes its local sorted chunk to disk, resulting in a fully sorted dataset split across multiple output files.

This design removes central bottlenecks and improves both load balancing and communication efficiency. All ranks contribute equally to sorting, and no node is overloaded. The final result is already partitioned, and no reconstruction or key remapping is required.

5.3 Implementation Differences Summary

Aspect	`par_merge`	`par_merge_psrs`
Data loading	Centralized (rank 0)	Distributed
Sorting target	Keys only	Full records
FastFlow merge	Sequential	Parallel (with feedback)
Inter-node merge	Tree-based (`MPI_Send/Recv`)	PSRS (`MPI_Alltoallv`)
Final array	Reconstructed at rank 0	Distributed across ranks
Output writing	Single output file	One output file per rank
Scalability	Limited by root	Scales linearly with ranks

Table 1: Comparison between the two hybrid implementations.

6 Experimental Evaluation

This section presents the experimental evaluation of the three implementations developed: the sequential baseline (`plain_sort`), the hybrid version with tree-based MPI merge (`par_merge`), and the PSRS-based hybrid version with distributed sorting (`par_merge_psrs`). The goal is to assess their performance in terms of scalability, speedup, and efficiency under various computational settings.

Test Environment

All experiments were performed on the `spmcluster` infrastructure. Each compute node is equipped with 16 physical cores (32 logical via hyper-threading) and interconnected via a high-speed network. Nodes access a shared file system managed by the front-end node, which is also used for compilation and submission.

Parameters and Configurations

The tests covered a broad range of configurations by varying:

- **Input size**: datasets of 10 million and 100 million records.
- **Payload size**: fixed-size payloads of 8, 64, and 256 bytes per record.
- **Parallelism**: number of MPI nodes and threads per node (FastFlow workers), with total concurrency ranging from 1 to 32.
- **Task distribution**: the number of MPI tasks per node, affecting load balance and communication patterns.

All datasets were pre-generated and loaded from binary files to ensure consistency and reproducibility across experiments.

Performance Metrics

We focus on the following performance indicators:

- **Strong Scaling Speedup**: how performance improves as the number of resources increases for a fixed problem size.
- **Weak Scaling Efficiency**: how well the implementation maintains performance when both problem size and resources grow proportionally.
- **Comparative Analysis**: between implementations for the same configuration to highlight trade-offs in communication, memory, and computation.

6.1 Speedup and Efficiency on a Single Node (Payload 64B)

To evaluate the performance of the proposed parallel sorting strategies, we measured the speedup and efficiency as the number of FastFlow threads increases on a single compute node. Each implementation was benchmarked on datasets of 10 million and 100 million records, using a payload size of 64 bytes. Speedup was computed against the runtime of a sequential baseline executed with `std::sort`.

Figures 1 and 2 show the measured speedup for both implementations. The results confirm a clear divergence in behavior.

The `par_merge` implementation demonstrates very limited scalability. On 10M records, speedup remains around 0.46 across all thread counts, indicating that the sequential bottleneck at the emitter—responsible for the final merge—completely dominates the parallel phase. Even with larger datasets (100M records), speedup stays below 0.47, confirming that increasing the problem size alone does not alleviate the inherent serialization bottleneck.

In contrast, the `psrs` implementation delivers nearly ideal behavior on the small input and performs even better on larger data. With 100M records, it achieves a speedup of over 5× at 32 threads. This improvement is due to two main factors: the removal of the emitter bottleneck through parallel merging, and the more balanced distribution of work across threads during both the sorting and merging phases.

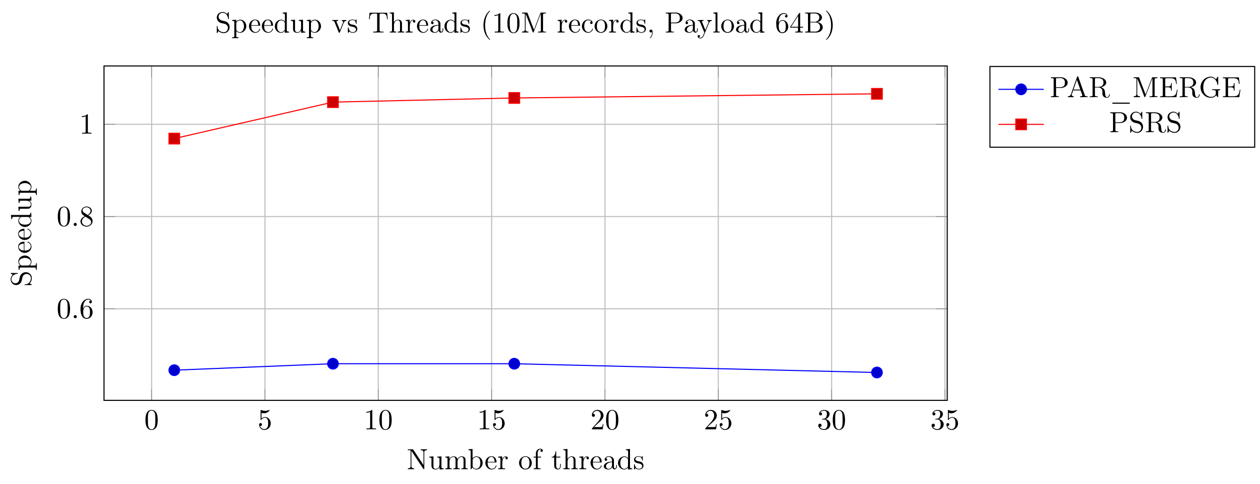

Figure 1: Measured speedup of PSRS and Tree-based Merge vs ideal linear speedup, on 10M records with payload 64B.

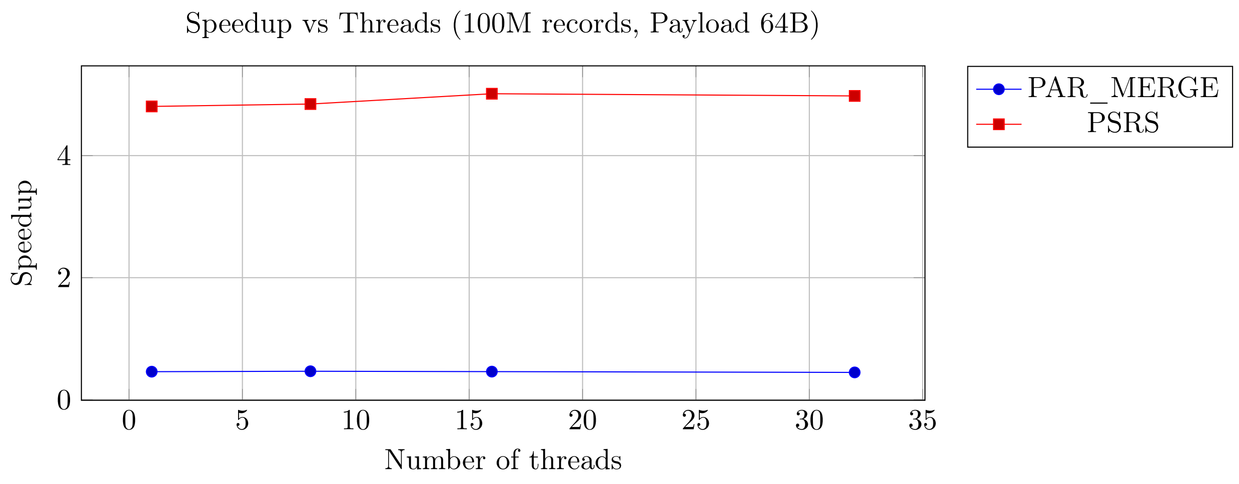

Figure 2: Measured speedup of PSRS and Tree-based Merge vs ideal linear speedup, on 100M records with payload 64B.

Figure 3 reports the efficiency trends for the 100M dataset. While `par_merge` suffers from an extreme drop in efficiency—falling below 2% at 32 threads—`psrs` maintains over 60% efficiency up to 8 threads and gracefully degrades beyond that, following a more expected pattern for memory-bound workloads.

These results show that the choice of merge strategy has a decisive impact on single-node scalability. The sequential nature of tree-based merging in `par_merge` severely limits its potential, while the parallel merge in `psrs`, combined with feedback-enabled task scheduling in FastFlow, enables consistent gains and a better use of system resources.

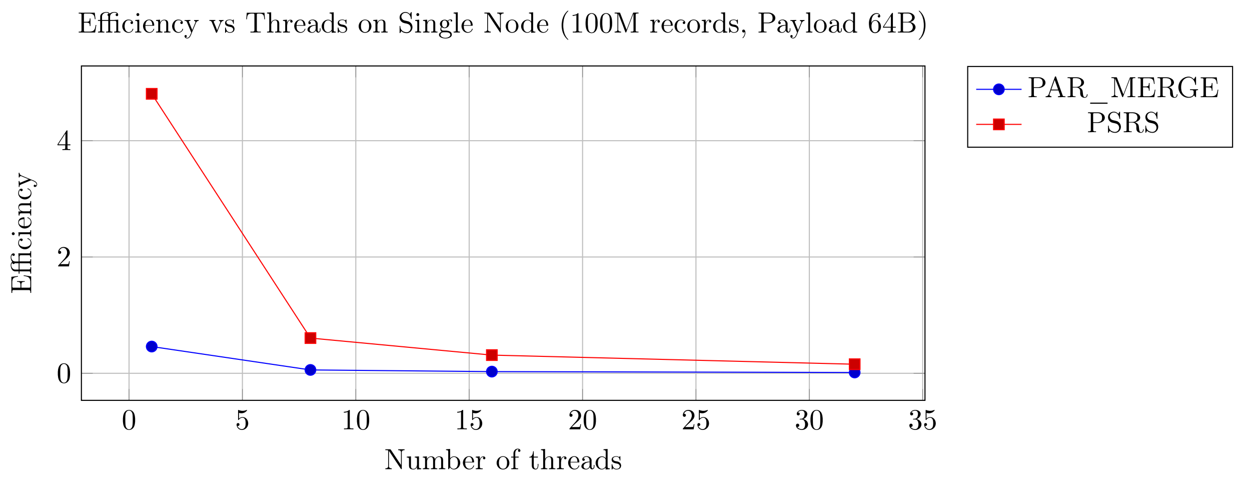

Figure 3: Parallel efficiency of PSRS and Tree-based Merge on 100M records with payload 64B.

It is also worth emphasizing that sorting in this context remains a memory-bound operation with limited opportunities for computational reuse. Since the payload is opaque and untouched during sorting, performance hinges entirely on memory access and data movement. As thread counts increase, contention for memory bandwidth and cache pressure grow, further limiting achievable speedup—even for well-balanced strategies like `psrs`. Nonetheless, the parallel design choices adopted in `psrs` allow it to amortize communication and coordination overheads more effectively than `par_merge`, making it a significantly more scalable solution in the shared-memory setting.

6.2 Strong Scaling Comparison Across Nodes (Payload 64 B)

To assess how well the two distributed algorithms scale when we add MPI nodes, we executed both `psrs` and `par_merge` on 100 million 64-byte records. Each run employed 32 OpenMP threads per node while the MPI node count took the values $\{1, 4, 8\}$. Speed-up (4)is reported with respect to the 16-thread runtime on one node, so values larger than one on a single node already include the benefit of intra-node parallelism; conversely, values below one indicate that the algorithm is slower than the baseline even before inter-node communication starts. By incorporating the parallel file-loading phase, the metric captures the full, end-to-end cost of the workflow.

On a single node, `psrs` enjoys an impressive five-fold acceleration, which reflects its ability to saturate memory bandwidth and overlap local partitions efficiently. Unfortunately, that advantage vanishes almost entirely once we distribute the workload: with four nodes the speed-up collapses to unity, and with eight nodes it climbs only to about 1.3 $\times$. The sharp drop stems from the algorithm's expensive pivot exchange, the global redistribution phase, and—in particular—the contention that arises when many ranks hit the shared file system simultaneously.

In stark contrast, `par_merge` never manages to outperform the single-thread baseline. Whether we run on one, four, or eight nodes, the speed-up lingers around 0.45 $\times$. The reason is twofold: first, the algorithm generates more I/O traffic per record than `psrs`; second, its final tree-merge step is essentially serial, so the cost of communication and synchronisation grows while the amount of useful parallel work remains fixed.

Taken together, the results underscore that neither approach comes close to the ideal linear trend. Once intra-node parallelism has been exploited, scaling is dictated by memory bandwidth, network latency, and, above all, the throughput of the shared storage subsystem. Beyond a single node, adding compute capacity does little except amplify those bottlenecks.

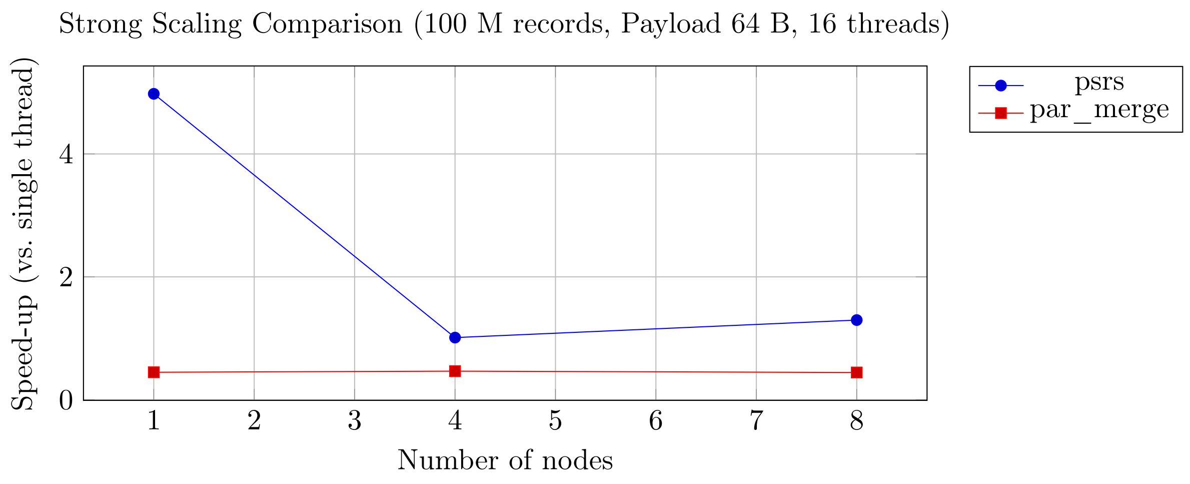

Figure 4: Strong-scaling speed-up for `psrs` and `par_merge`.

6.3 Weak Scaling Analysis (Payload 64)

To assess the weak scalability of the proposed sorting strategies, we fix the per-node workload while increasing the number of nodes. Each node is assigned 8 threads and a proportional amount of data: 20 million records on 1 node, 80 million on 4 nodes, and 160 million on 8 nodes. If the implementation scales well, the overall execution time should remain roughly constant despite the growing total problem size.

Figure 5 presents the total execution time (including both data loading and sorting) for the two implementations: `par_merge` and `psrs`. The `psrs` implementation demonstrates excellent scalability, with execution time increasing sub-linearly as the number of nodes grows. This confirms the effectiveness of parallel merge and regular sampling in distributing work evenly and minimizing bottlenecks.

In contrast, `par_merge` exhibits significant deterioration in execution time. The sequential merge performed at rank 0 and the key-based reconstruction of the final result become dominant costs, especially as the problem size increases. Communication and coordination overheads are not effectively amortized, leading to poor scaling.

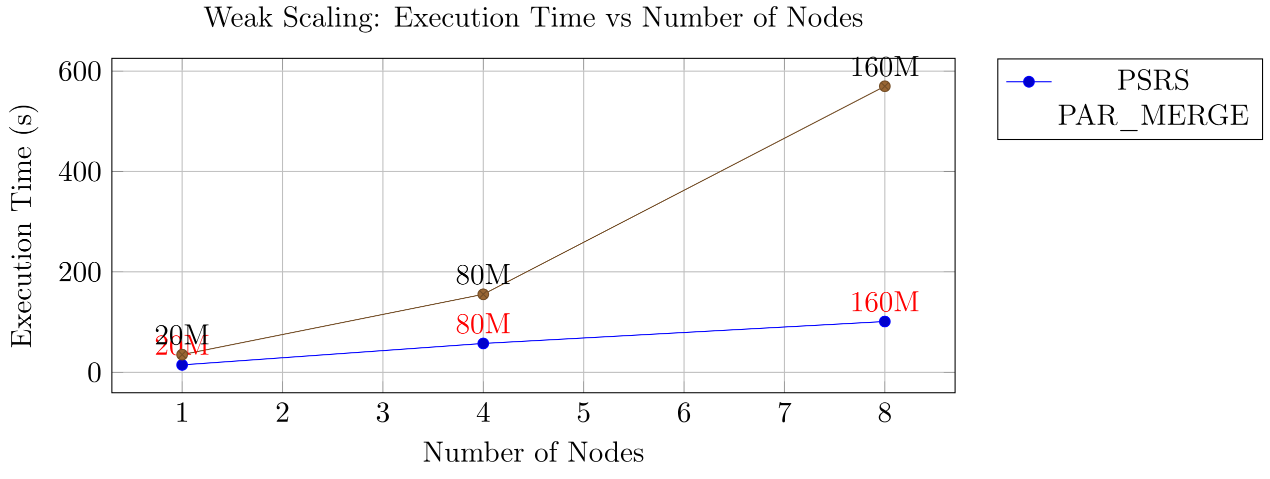

Figure 5: Total execution time under weak scaling conditions with 8 threads per node (including get, sort, write).

To quantify the trend, Figure 6 reports **Weak Scaling Efficiency**, computed as:

$$E_p = \frac{T(1, N)}{T(p, p \times N)}$$

where $T(1, N)$ is the execution time required to sort N records on a single node using 8 threads, and $T(p, p \times N)$ is the execution time to sort $p \times N$ records on p nodes, again using 8 threads per node. This definition captures how well the total execution time holds up as both the dataset size and the number of nodes scale proportionally.

Ideally, the ratio remains close to 1, meaning the system handles larger problems without increasing execution time. As shown in Figure 6, both implementations suffer from sharp degradation.

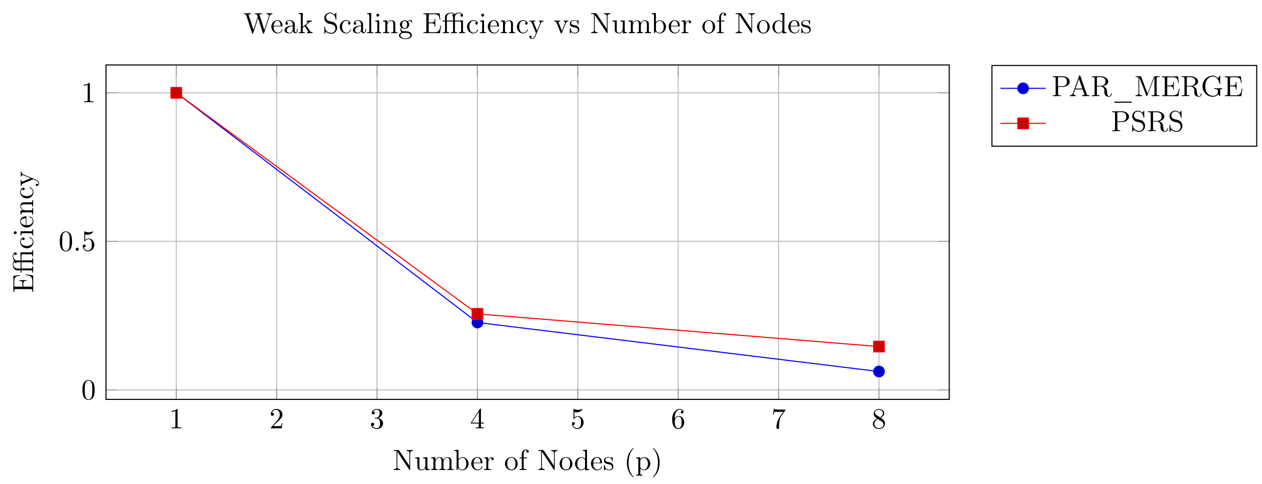

Figure 6: Weak scaling efficiency computed as $E_p = \frac{T(1,N)}{T(p,p\times N)}$, showing stronger scalability of PSRS.

7 Challenges and Design Trade-offs

The development of this project highlighted several challenges and trade-offs, both in terms of algorithmic design and practical implementation. The main difficulty stems from the nature of the problem itself: sorting fixed-size opaque records without any computation on the payload. This makes the entire process inherently memory-bound, limiting the potential gains from parallel execution.

Limited Computational Leverage

Unlike numerical simulations or compute-heavy kernels, sorting opaque records offers little room for arithmetic optimization or SIMD acceleration. The only available parallelism comes from data partitioning and concurrent comparison/swap operations. Moreover, once the keys are sorted, payload reordering becomes a purely memory-bound task. This explains why speedup curves remain modest and often sublinear, even when leveraging 32 threads or 8 nodes.

Communication vs Computation Overlap

Particular care was taken to overlap computation and communication whenever possible. In the `par_merge` version, communication is lightweight (only keys are exchanged), but the merge step is centralized at rank 0, introducing a bottleneck. On the contrary, in the `psrs` version, each rank sends and receives full `Records`. While this increases communication volume, it occurs fewer times and in a more structured fashion (via `MPI_Alltoallv`), and each rank ends up independently responsible for a portion of the globally sorted array.

This trade-off—between the number of messages and the volume per message—was fundamental in the scalability difference observed between the two versions.

FastFlow: Ease of Implementation vs Performance

On the intra-node side, two strategies were explored for the merge phase:

- A sequential merge performed by the emitter after all workers sorted their partitions. This is easy to implement and minimizes synchronization, but becomes a bottleneck as the number of threads grows.
- A parallel merge with FastFlow feedback channels, where workers are dynamically reused to merge results in a tree-like fashion. This design is more complex but improves scalability and balances load across all threads.

The choice of the second strategy in the final PSRS version allowed better intra-node performance, at the cost of higher implementation effort.

Memory Bottlenecks and Load Distribution

A key difficulty in the `par_merge` version was the centralized nature of memory management: rank 0 had to receive and reconstruct the full sorted array. This introduced severe memory pressure and limited scalability. In contrast, the PSRS strategy ensured even memory usage across all nodes, since each rank handled only a slice of the final result.

Moreover, the distributed sampling and splitter-based partitioning in PSRS provided a more balanced division of work, especially when record keys were uniformly distributed.

MPI and I/O Considerations

Parallel file loading from the shared filesystem also emerged as a limiting factor, especially when increasing the number of nodes. Since I/O occurs concurrently across processes, contention on the filesystem can quickly dominate execution time. This was particularly evident in the strong and weak scaling experiments, where scalability plateaued or degraded due to shared I/O bottlenecks.

Final Thoughts

Overall, the project involved carefully navigating multiple trade-offs:

- Between centralized simplicity and distributed scalability.
- Between low-volume frequent communication and high-volume structured communication.
- Between easy-to-implement sequential merges and more scalable—but complex—parallel merges.

Despite the inherent limits imposed by the problem domain, the final design of the `par_merge_psrs` version achieved good scalability and a clean separation of responsibilities between MPI and FastFlow. This allowed for modular development, reproducible experiments, and consistent behavior across datasets of up to 160 million records.

8 Conclusions and Future Work

This project provided the opportunity to design, implement, and evaluate multiple parallel sorting strategies in a realistic distributed-memory environment. Starting from a sequential baseline using `std::sort`, we progressively moved towards more complex and scalable solutions by integrating FastFlow for intra-node parallelism and MPI for inter-node coordination.

The two hybrid implementations explored different trade-offs. The `par_merge` version focused on minimizing communication by distributing only the keys and performing a centralized merge at rank 0. While this solution was easier to implement, its scalability proved to be limited, primarily due to the sequential merge bottleneck and the memory pressure placed on a single process.

In contrast, the PSRS-based implementation (`par_merge_psrs`) showed better scalability by fully distributing both data and computation. Thanks to a parallel merge strategy within nodes and a well-structured global partitioning phase, each rank independently produces a sorted portion of the final array. This eliminates the need for central reconstruction and results in a much more scalable architecture.

Still, as observed in the experimental section, the performance gains remained modest. This is largely due to the nature of the problem: sorting opaque records does not involve any real computation, and the task is dominated by memory access and data movement. In this context, parallelism helps, but cannot fully eliminate the bottlenecks imposed by synchronization and communication.

Future Directions

There are several directions in which the current implementation could be improved. One important aspect is the overlapping of communication and computation. In the PSRS version, we already try to pipeline sampling and redistribution with local sorting, but more could be done using non-blocking MPI primitives to reduce idle time during data exchange.

Another improvement concerns the FastFlow implementation itself. While the parallel merge with feedback channel improves scalability, it could be made even more efficient by optimizing memory allocation, improving cache locality, or reducing the number of temporary buffers used during merge phases.

The way payloads are handled could also be revisited. Currently, payloads are treated as opaque blocks of memory and moved around without any processing. In more advanced scenarios—e.g., when the payloads are compressible or partially discardable—additional gains could be achieved by reducing the amount of data exchanged between nodes.

Finally, while our current PSRS strategy assumes a roughly uniform distribution of keys, many real-world datasets exhibit skewed distributions. In such cases, the regular sampling step might produce unbalanced partitions. Future versions could adopt adaptive sampling or histogram-based techniques to improve load balancing across ranks.

In summary, although the problem tackled is theoretically simple, achieving efficient performance in a real distributed system required careful design choices and highlighted the trade-offs between ease of implementation, communication overhead, and memory usage. The PSRS-based implementation lays a strong foundation for further experimentation and extension.