Assignment 4: Hybrid MPI + FastFlow MergeSort

Luca Lombardo
SPM Course a.a. 24/25

The goal of this project was to develop a scalable MergeSort algorithm for a large array of records, combining intra-node parallelism with FastFlow and inter-node parallelism with MPI.

1 Implementation Strategy

At the outset, a key design decision was made regarding the `Record` data structure. While the assignment specified a fixed-size payload array (`char rpayload[RPAYLOAD]`), we opted for a more flexible structure with a dynamically allocated payload (`char* payload`). This choice was motivated by the need for a versatile and robust benchmarking framework. Using a dynamic payload allows the record size to be configured at runtime via command-line arguments, which is very useful for systematic performance analysis across various payload sizes without requiring code recompilation for each test. This design choice does not compromise the performance analysis. The memory for each record's payload is allocated once at the beginning of the tests and remains fixed throughout the sorting process. Within the core sorting algorithms, the cost of moving or comparing these records is dominated by the size of the payload. While this approach introduces a single level of pointer indirection compared to a static array, its impact is negligible for a memory-bandwidth-bound streaming algorithm like MergeSort. The use of move semantics ensures that during merge operations, only the pointer and size are transferred, preserving performance characteristics equivalent to those of a fixed-size struct.

1.1 Single-Node Parallel Design (FastFlow)

A classic MergeSort naturally decomposes into two distinct phases: an initial, highly parallel sorting of small data chunks, followed by a series of sequential merge passes. Our implementation mirrors this structure, employing dedicated FastFlow farms for each phase to optimize resource utilization.

The first phase, parallel sorting, addresses the initial data partitioning. The array is divided into numerous small, cache-friendly chunks to maximize data locality. The sorting of these independent chunks represents an embarrassingly parallel problem, for which we employed an `ff_farm`. An emitter node decomposes the array into tasks which are distributed to a pool of worker threads. Each worker leverages `std::sort` to sort its assigned chunk in-place.

The second phase, iterative merging, progressively combines the sorted chunks. This process is inherently iterative and is implemented with a loop where, at each iteration, a new `ff_farm` executes a single merge pass in parallel. A critical optimization in this stage is the use of a "ping-pong" buffer strategy. By allocating a single auxiliary buffer and swapping pointers between source and destination at the end of each pass, we eliminate the significant overhead of copying data between merge stages. This out-of-place merge technique is standard for maximizing memory throughput. The merge operation itself is optimized through `std::make_move_iterator`, ensuring that for our `Record` structure, only pointers are moved, not the potentially large payloads, thus minimizing data transfer costs on the memory bus.

1.1.1 Architectural Considerations and Discarded Alternatives

The assignment suggested exploring FastFlow's building blocks, including `ff_pipeline` and `ff_all2all`. We evaluated these alternatives but ultimately discarded them for sound performance and algorithmic reasons.

The `ff_all2all` pattern is designed for data redistribution, where each worker sends a piece of its local data to every other worker. This communication pattern is fundamental to algorithms like Radix Sort or Sample Sort but is algorithmically incompatible with MergeSort, which relies on merging physically adjacent, sorted segments in a hierarchical, pairwise fashion.

An integrated pipeline architecture, such as `feeder -> sort_stage -> merge_stage`, also proved suboptimal. Our benchmarks showed significant performance degradation. This is because MergeSort is fundamentally a Bulk-Synchronous Parallel (BSP) algorithm. There is a hard synchronization point after the initial sort phase; the merge phase cannot begin until all chunks are sorted, precluding any opportunity for true, overlapping pipelining. The inter-stage communication and scheduling overhead introduced by the `ff_pipeline` outweighs any potential benefits compared to the lean approach of dedicating all parallel resources to a single, optimized farm for each distinct computational phase.

1.2 Hybrid Multi-Node Design (MPI + FastFlow)

To scale the algorithm beyond a single node, we designed a hybrid implementation that combines MPI for inter-node coordination with our optimized FastFlow MergeSort for intra-node computation. The overall process is structured into three distinct phases.

The first phase is a load-balanced data distribution. The root process (rank 0) partitions the global dataset of size N into P contiguous blocks, where P is the MPI world size. To handle cases where N is not perfectly divisible by P, we calculate unique per-process element counts and memory displacements. This information is then used with the collective operation `MPI_Scatterv`, which is more general than a simple `MPI_Scatter` and ensures an even distribution of records. For efficiency, especially with non-trivial payloads, data is first serialized into a contiguous byte buffer. This strategy minimizes the number of discrete MPI calls and optimizes for network throughput by sending a single, large message chunk to each process.

The second phase is local sorting. Upon receiving its data partition, each MPI process executes the multi-threaded parallel mergesort developed before. This step performs a coarse-grained, computation-bound sort on the local data, leveraging all available cores on the node as configured. This transforms the global problem into a set of P distributed, independently sorted arrays, preparing the data for the final merge phase.

The third and most complex phase is hierarchical merging. After local sorting, the distributed segments must be merged into a single, globally sorted array at the root. We chose a binary tree reduction pattern for this task, which completes in $\log_2(P)$ communication steps. At each step, active processes are paired: a "sender" transmits its data to a "receiver," which then performs a two-way merge. This pattern effectively distributes the merge workload across multiple nodes in the early stages, but progressively serializes the work toward the root process, which performs the final merge.

A critical aspect of this phase is the communication strategy, designed to maximize computation-to-communication overlap. Our implementation employs a non-blocking model. Each receiving process determines all future merge partners upfront and posts non-blocking receives (`MPI_Irecv`) for all of them immediately. It then enters a progress loop, using `MPI_Waitany` to react as soon as any one of these data transfers completes. Upon completion, the received data is immediately processed and merged. This event-driven approach allows the computational cost of merging one data block to overlap with the network latency of other pending transfers, significantly improving resource utilization and hiding communication overhead.

1.2.1 Alternative Merge Strategy: Centralized K-Way Merge

We also evaluated an alternative strategy for the final merge phase: a centralized k-way merge. In this model, all $P-1$ non-root processes send their locally sorted data directly to the root process, which then performs a single, large k-way merge using a min-heap data structure.

Theoretically, this approach is attractive as it reduces the number of communication rounds from $\log_2(P)$ to one. However, our benchmarks showed it to be less performant than the binary tree reduction. The primary reason is that it creates a massive communication and computation bottleneck at the root process. The root must handle incoming network traffic from all other processes simultaneously while also performing the entire merge. This serializes a large fraction of the work and leaves the other nodes idle after their initial send. In contrast, the binary tree approach distributes the merge workload across multiple nodes in the initial steps, leveraging the computational power of the entire system for a longer duration.

2 Performance Analysis

We benchmarked our implementations to evaluate their strong and weak scaling properties. The analysis is based on the data collected and visualized below.

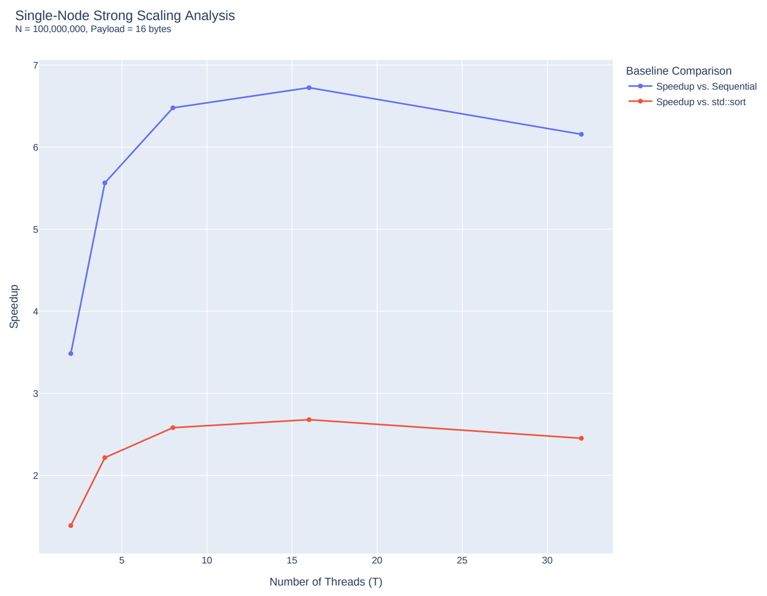
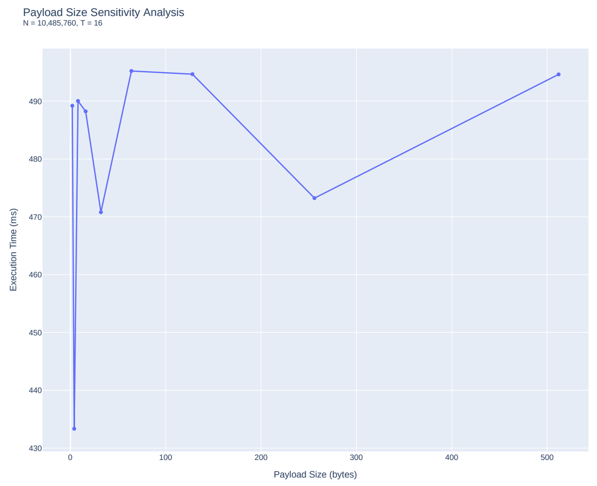
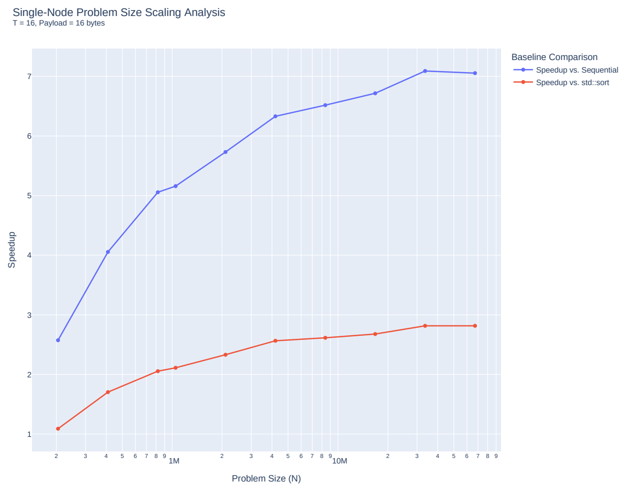


Figure 1: Single-node strong scaling analysis.

Figure 1 shows the strong scaling of our FastFlow implementation. The speedup increases significantly up to 16 threads, reaching a peak of approximately 6.8x against our sequential implementation. Beyond this point, performance saturates and then begins to degrade. This behavior is an expected illustration of Amdahl's Law, where the parallel overhead, including thread management, synchronization, and memory bus contention, becomes a dominant factor. As the number of threads grows, the marginal gain from additional parallelism diminishes until it is negated by these overheads, defining an optimal level of concurrency for this specific problem size and architecture.



(a) Payload size sensitivity analysis.

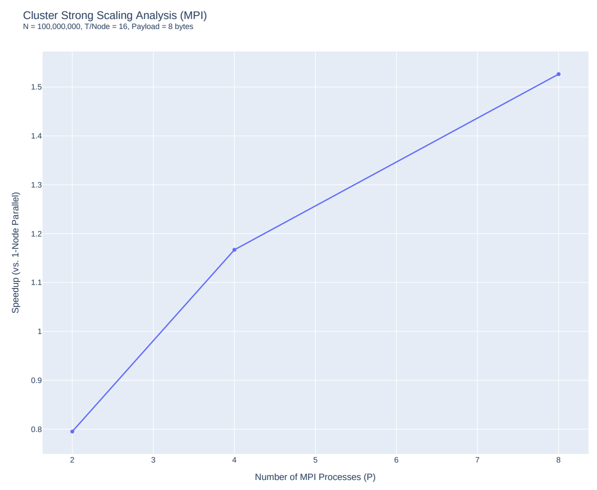


(b) Single-node problem size scaling.

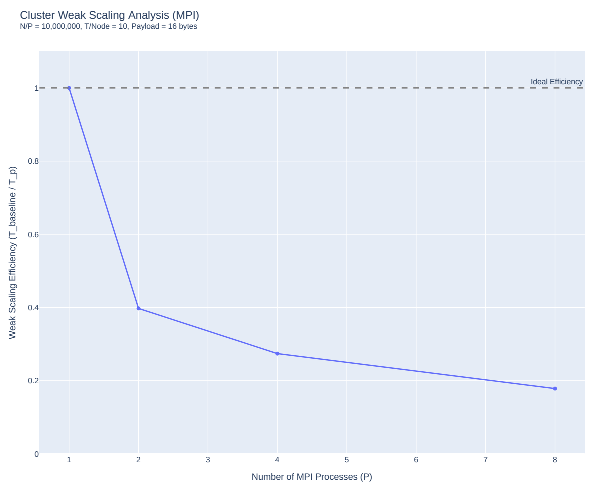
Figure 2: Single-node performance sensitivity to payload and problem size.

The impact of payload size is shown in Figure 2a. The execution time remains relatively stable for payloads from 8 to 512 bytes. This indicates that the algorithm's performance is heavily influenced by the system's memory bandwidth. The primary cost in the merge phase is data movement, which scales linearly with the record size. As the payload grows, the problem transitions from being potentially latency-sensitive to firmly memory-bandwidth-bound. The time spent in CPU-bound computation (key comparisons) becomes negligible compared to the time the memory subsystem spends satisfying data transfer requests.

Figure 2b presents a scaling analysis where the number of threads is fixed, and the total problem size is increased. It is important to note that the x-axis (Problem Size) is plotted on a logarithmic scale. The speedup exhibits strong growth with the problem size, appearing almost linear on this log-scale plot, which indicates a highly favorable scaling behavior. This aligns with the principles of Gustafson's Law: by increasing the problem size, the serial fraction of the work becomes less significant relative to the parallelizable portion. With more data to process, the initial fixed costs of setting up the FastFlow framework and threads are amortized over a longer execution time, leading to substantially better efficiency and speedup.



(a) Cluster strong scaling (MPI).



(b) Cluster weak scaling (MPI).

Figure 3: Distributed performance analysis: strong and weak scaling on the cluster.

Figure 3a provides an instructive view into the strong scaling characteristics of the hybrid algorithm. The speedup is measured relative to an optimized single-node parallel baseline (i.e., the same algorithm running on one MPI process with the same number of FastFlow threads). The results show a successful scaling behavior. While the two-process execution is slower than the baseline (speedup of approx. 0.8) due to the initial overhead of network communication, the implementation achieves a significant speedup with four processes (approx. 1.17x) and scales further to eight processes (approx. 1.53x). This demonstrates that for a large problem size of 100 million records, the benefits of distributing the computational load across multiple nodes effectively overcome the costs of data transfer and synchronization. The non-blocking communication strategy, which overlaps merge operations with data transfers, proves crucial in mitigating network latency. The diminishing efficiency returns as more nodes are added are expected in strong scaling scenarios, as communication costs inevitably begin to occupy a larger fraction of the total execution time, in accordance with Amdahl's Law.

To complete the performance study, we conducted a weak scaling test on the cluster, where the problem size per node was held constant. Figure 3b plots the resulting efficiency, where an ideal algorithm would maintain a value of 1.0. Our results demonstrate a sharp degradation in efficiency: from perfect efficiency at one node, it drops to approximately 0.4 at two nodes, and continues to decline to below 0.2 with eight nodes. This behavior is characteristic of algorithms with non-scalable communication patterns, such as distributed MergeSort. The primary cause for this poor weak scaling lies in the hierarchical merge phase. While the local sort phase exhibits perfect weak scaling, the communication complexity of the binary tree reduction increases with the number of processes P. In each of the $\log_2 P$ steps, processes must transmit increasingly large, already-merged partitions. The total volume of data communicated and the latency of the final, more sequential merge steps grow with P.