Fourth Assignment - Parallel merge sort in SM and DM

SPM Course 2024/2025

[Github repo]

Student: Luca Rizzo

ID Number: 598992

Problem Overview and Parallelization Strategy

The problem addressed in this project is the parallelization of the MergeSort algorithm, a classic divide-and-conquer sorting technique. The goal is to efficiently sort a large array of records, leveraging both **shared-memory** and **distributed-memory** architectures

In the case of MergeSort, the algorithm's cost is classically analyzed in terms of the number of element comparisons. However, since comparisons dominate the work performed, especially in large inputs, this measure can be reasonably approximated as the overall computational cost. In this sense, the total cost for non-base cases can be expressed as `T(n) = D(n) + T(n/2) + M(n)` where:

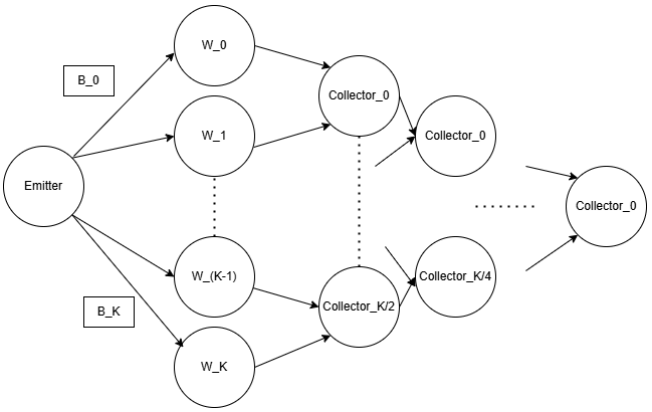
- `D(n)` is the cost of the divide phase (typically negligible, O(1)),
- `M(n)` is the cost of the merge phase, which dominates with a complexity of O(n).

Given this cost structure, it's evident that the merge step is the most critical and expensive phase to parallelize. For this reason, both our shared-memory implementation using FastFlow and the distributed-memory implementation using MPI have been designed with the primary goal of optimizing the merge phase. To do so, both versions adopt a **tree-based hierarchical merging strategy**, which enables scalable and structured parallel execution. The divide phase, that is essentially a scatter, being negligible in cost, is kept simple in our implementation, allowing us to focus optimization efforts entirely on the true computational bottleneck: the merge phase.

Shared memory

Implementation analysis

The entire sorting procedure can be conceptually framed as a **Map with a tree-based Gather**, a classic pattern in data parallel programming. In this model, the input collection is first split into multiple independent chunks, each of which is sorted separately and once all chunks are sorted, the algorithm proceeds with a hierarchical merge phase, which can be seen as a structured Gather. This phase resembles a tree-reduction pattern, where sorted subarrays are merged level by level, progressively reducing the number of segments until a single fully sorted array remains. A diagram that represents the logical view of processing is shown alongside.

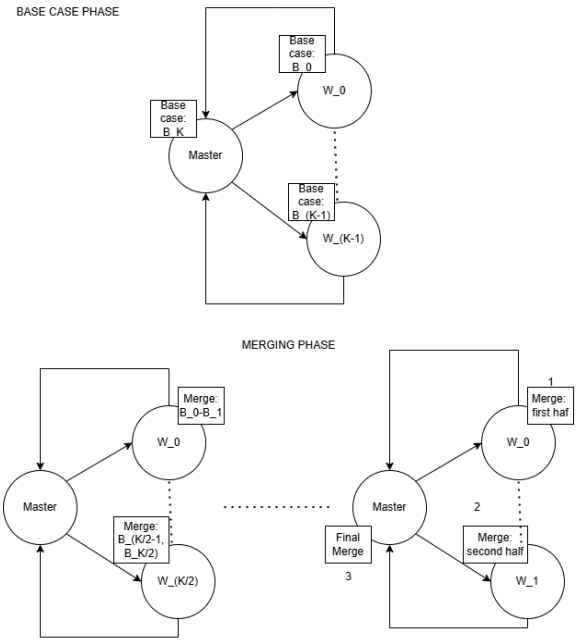


This conceptual model is implemented using a classic **Master-Worker** pattern, where the Emitter acts as the master and the Workers perform both sorting and merging tasks. The implementation is based on the `ff_MergeSort_Map` class, which extends FastFlow's `ff_Farm`. This class encapsulates the internal allocation and coordination logic required to perform a parallel merge sort. As the user provides the total number of FastFlow threads via the `-t` option (`par_degree`), the farm constructor initializes `par_degree-1` workers, as one thread is already used by the master (emitter). Since we are in a shared-memory system, both the master and the workers operate directly on a common data buffer using a pointer. Tasks are defined by index ranges `[begin, end)`, allowing workers to process their assigned segments in-place without copying data. Communication and synchronization are handled

implicitly via FastFlow's task queues. There is no overlap or race condition, as each worker always operates on disjoint portions of the array, whether performing base case sorting or merging.

Initially, the master partitions the input collection into multiple base cases and dispatches them to the workers as sorting tasks. One base case is retained and sorted locally by the master. By default, the size of each base case is computed as $\lceil numRecords/(numFFThreads) \rceil$, ensuring an even distribution of work. However, this granularity can be customized at runtime using the `-b` command-line option, which allows the user to manually set the base case size.

Beyond this initial distribution, the master coordinates the entire **hierarchical merge phase**, emulating the Gather step level by level using a bottom-up merge strategy. Each task completion acts as a notification from a worker. Once all current-level tasks are done, the master generates and dispatches merge tasks for the next level.



At merge level 0, the master assigns the merge of blocks B0 and B1 to the first worker, B2 and B3 to the second, and so on. Once all merge tasks at the current level are completed, the master schedules the merges for the next level, for example, merging B0–B3 as a single unit and assigning it to the first worker. This iterative process continues until a single, completely sorted block remains. When the master receives the final two sorted blocks, it performs the last merge itself instead of dispatching a new task to the workers. This avoids unnecessary overhead, as the workers have already completed their tasks and the master is idle at that point. For the base case sorting tasks, the standard C++ library function `std::sort` was used, instead for the merging phase, `std::inplace_merge` was adopted to perform in-place merging of sorted subarrays.

Since base case sorting tasks operate on randomly distributed, unsorted data blocks, and merge tasks combine sorted blocks of similar size, there is no significant load imbalance among them on average. For this reason, a **block-cyclic distribution** of tasks across workers is expected to perform well, as it ensures a regular distribution while still maintaining flexibility in workload assignment. This scheduling strategy promotes good load balancing without requiring dynamic work stealing or runtime adaptation.

To optimize resource usage, if the number of active workers exceeds the number of available merge tasks at a given level, the master sends EOS (End Of Stream) signals to excess workers to proactively terminate them and release system resources.

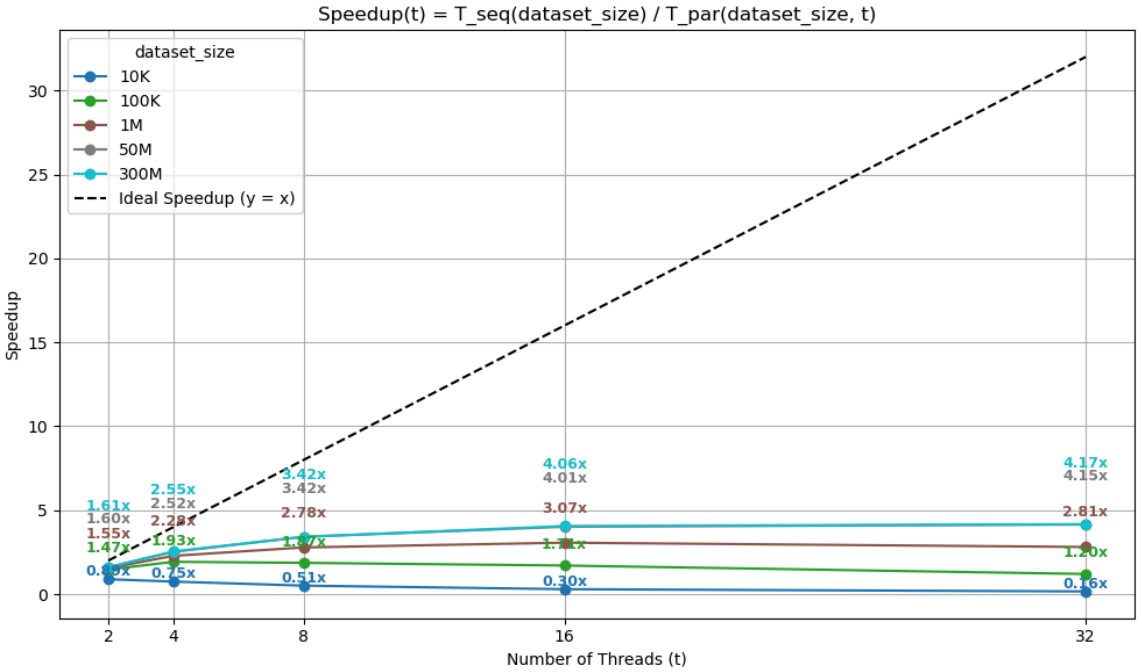
Performance analysis

Since the implementation does not manipulate or access the actual contents of the payloads, storing only the pointers within each `Record`, the `record_payload_size` parameter (`-r`) has been kept constant at 5 byte throughout the reported experiments. Although additional tests were conducted with varying payload sizes, the observed performance remained largely unaffected. To keep the discussion focused and concise, those additional results have been omitted from this report.

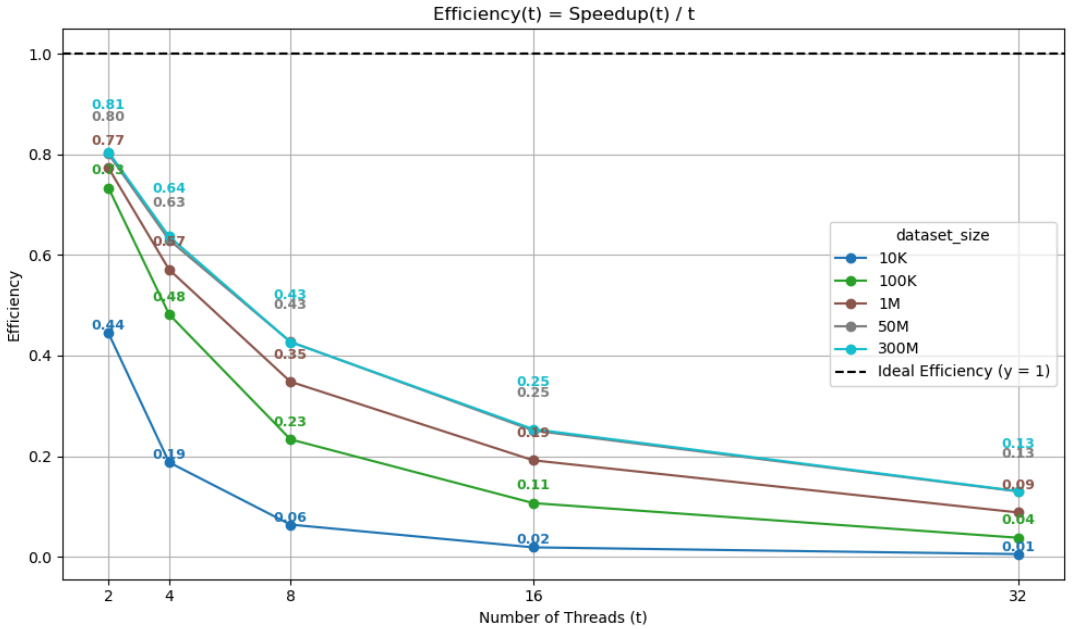
Speedup and efficiency

As a sequential baseline, the standard C++ `std::sort` function was used to sort the entire input array on a single thread. This served as a robust and optimized reference for evaluating the parallel implementation. Speedup was then measured for various dataset sizes using different thread counts. The results show that performance benefits are clearly noticeable starting from datasets of around 100K elements, where the overhead introduced by task management and thread coordination in FastFlow is sufficiently amortized. For example, at 100K, a speedup of approximately 2× is achieved with 4 threads. For larger datasets, the efficiency improves further: in particular, for 300M elements, the speedup reaches around **4.17×**, demonstrating that the parallel workload becomes increasingly effective as the problem size grows. On the other hand, for very small datasets such as 10K, the parallel overhead outweighs the computational gain, resulting in suboptimal speedup. This highlights that parallelization becomes effective only when the problem size is large enough to justify the cost of coordination and scheduling.

While these results are still far from the ideal linear speedup, they are nonetheless encouraging. One of the limiting factors lies in the **final merging steps**, which are **inherently sequential** operations: as the merging progresses toward a single final output, only one thread can complete the final combination. This represents a non-parallelizable portion of the algorithm, which is structurally necessary and unavoidable. In the context of Amdahl's Law, this corresponds to the serial fraction of the computation, and it ultimately places an upper bound on the achievable speedup, regardless of how many threads are used.



This effect is further reflected in the efficiency plot, which naturally decreases as the number of threads increases. This is expected, as we are not able to fully utilize all threads throughout the computation. In particular, during the final levels of the merge phase, some workers are intentionally terminated (via EOS) when they are no longer needed, leading to **partial underutilization** of the available resources.



Distributed memory implementation

Implementation analysis

In the distributed version of the algorithm, we extend the shared-memory design to a cluster of nodes using MPI. The goal is to parallelize the sorting process across multiple processes, each running on a separate node of the cluster, while maintaining scalability and performance.

In this implementation, I deliberately chose to adopt a strategy in which only process 0 obtains the entire sorted array at the end of the computation. This simplifies the final collection and avoids unnecessary data duplication across nodes. An all-reduce variant of the algorithm, where all processes obtain the complete sorted result, could still be implemented by adding a final broadcast step or by using a recursive doubling algorithm, as introduced in lecture.

Another design simplification was to **require the number of MPI processes to be a power of two**, in order to simplify the implementation of the hierarchical merge across levels. This ensures that the merging tree is perfectly balanced and avoids dealing with irregular cases during communication.

If the user provides a number of processes that is not a power of two, the program prints a warning message to the console and excludes the extra processes from the computation. Specifically, only the first `2^k` processes (where `2^k` is the largest power of two less than or equal to the total number of processes) participate in the sorting and merging phases.

To handle this cleanly, a new MPI communicator (`MPI_ACTIVE_COMM`) is created that includes only the active processes, using `MPI_Comm_split`. The excluded processes finalize and exit immediately, while the active ones proceed with the algorithm using the newly created communicator.

At the beginning of execution, each MPI process parses the command-line arguments to configure its local parameters. However, only the process with rank 0 is responsible for generating the full input array.

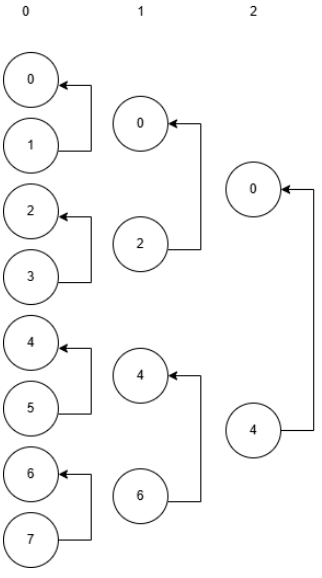
A crucial implementation detail is the use of a custom MPI datatype to represent the `Record` structure. While the `key` field is a standard 32-bit unsigned integer, the `payload` is a pointer, which has no meaning outside the process that allocated it. To reduce communication overhead, I deliberately chose **not to transmit the actual payload contents**, but only the pointer values. These pointers are treated as 64-bit integers and sent as-is to the other nodes, serving solely to preserve key–payload associations during the global sort.

This means that the payloads received by non-root nodes are not valid memory addresses, they are simply placeholders. However, since the sorting is performed only on the key, and the original pointers are preserved throughout the process, the final sorted array, which is reconstructed entirely by process 0, can transparently recover the correct key–payload associations. Passing only the keys would have made this association difficult to reconstruct, whereas passing the full `Record` with raw pointer values allows the sorting logic to remain consistent and centralized.

At a high level, the distributed MergeSort follows three main phases:

1. **Data Distribution (Scatter)**:
   The root process generates the input array and distributes it evenly across all active MPI processes using `MPI_Scatterv`. Each process receives a disjoint chunk of the array.
2. **Local Sorting (Map)**:
   Once each process has received its subarray, it locally sorts its chunk using the same FastFlow-based parallel MergeSort used in the shared-memory version. This allows each node to exploit its **internal parallelism**.
3. **Hierarchical Merge (Gather)**:
   After local sorting, processes are organized in a **binary tree structure** to perform the merge in parallel. At each level of the tree, a receiver process waits for data from its paired sender for that level, performs a local merge, and proceeds to the next level. The process continues until the root collects and merges all data into a globally sorted array.

In particular, at each merge level, a process takes on the role of either sender or receiver, based on its rank and the current level. A process acts as a sender if it provides its sorted chunk to another process, and as a receiver if it waits to receive data from a peer and performs a merge. Once a process completes its role as a receiver at a given level, it continues to the next level of merging. In contrast, senders terminate their activity after sending their data, as they are no longer part of the active computation.



This role assignment is structured to ensure that the merging process progressively reduces the sorted data toward process 0, which ultimately holds the fully sorted array. The hierarchy is constructed as a balanced binary tree, allowing efficient and scalable communication patterns.

A communication diagram illustrating the communication between 8 nodes during each merging level is shown alongside.

Since the tree structure used for merging is fixed and fully determined by the ranks and the number of processes, each node knows in advance up to which level it will act as a receiver, as well as the identities of its corresponding senders. It also knows the precise offsets in the buffer where incoming sorted segments will be written at each level. This knowledge allows each receiver to immediately post all asynchronous `MPI_Irecv` calls for all expected merge levels, even before beginning its local sorting

phase. As a result, sorted segments from other nodes can start arriving while the local sorting is still in progress, effectively **overlapping communication and computation**. The merging steps are performed in-place on a preallocated buffer large enough to hold the full set of segments a node will process, eliminating the need for repeated allocations and reducing memory overhead. For example, in a configuration with 8 MPI processes, **node 0**, which is the root of the merging tree, knows it will act as a receiver at every level. Consequently, it must be prepared to receive and merge all sorted segments, and therefore allocates a buffer large enough to hold the entire input array.

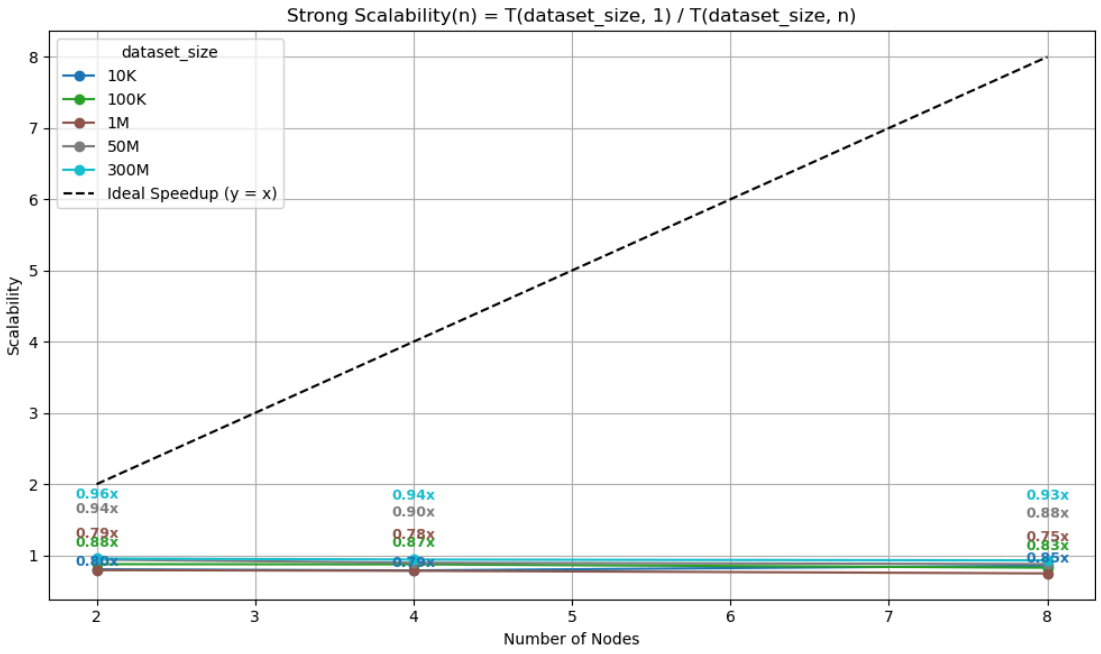
In contrast, node 4 participates only up to level 2 of the tree. It knows it will receive data at level 0 from node 5, and at level 1 from node 6 which in turn have already received and merged data from node 7. As a result, node 4's buffer only needs to accommodate half of the total input, corresponding to the maximum size it will ever be responsible for during the merging process.

Performance analysis

It is worth noting that, compared to the sequential execution on a dataset of 300M record with `std::sort` (with times of approximately 26.97 s and 26.83 s), the distributed implementation with 4 and 8 nodes achieves a total execution time of around 12 s (see later). This corresponds to a **speedup of more than 2×**, confirming that the distributed approach, despite its communication overhead, still outperforms the sequential baseline when the workload is large enough.

Strong scalability

To evaluate the strong scalability of the distributed implementation, I conducted a series of experiments by varying the number of MPI nodes among 2, 4, and 8, while keeping the dataset size fixed. Tests were repeated for different array sizes, from small collections (10K records) to very large datasets (300 million records). The resulting scalability curves are shown in the figure.



Strong scalability is defined as: `Strong_scalability(n) = T(dataset_size, 1) / T(dataset_size, n)` where `T(dataset_size, n)` is the total execution time when using `n` MPI nodes. As observed in the graph, the algorithm does not scale ideally. Even for large datasets, the

scalability remains well below the ideal linear speedup (represented by the dashed line `y = x`). In the case of the 300M dataset, some detailed runtime logs help us better understand what happens internally. These logs, shown in the figure, report the timing of the main phases of the algorithm (scatter, base case sorting, and merge) and allow us to identify where the primary **performance bottlenecks** occur.

Nodes	Scatter Time	Base Case Time	Merge Time	Base + Merge	Total Time
1	~1.45 s	~9.75 s	0.00 s	~9.75 s	~11.2 s
2	~2.76 s	~4.9 s	~4.15 s	~9.05	~11.8 s
4	~3.42 s	~2.43 s	~6.2 s	~8,63	~12.1 s
8	~3.76 s	~1.28 s	~7.2 s	~8,48	~12.2 s

These results clearly show that, while the base case sorting time is effectively reduced thanks to the parallelization inside each node, the overall execution time remains roughly constant or even increases slightly as more nodes are added. This is due to two dominant factors:

1. **Scatter Overhead**: The `MPI_Scatterv` phase becomes increasingly costly as the number of nodes grows, due to both data volume and network coordination.

2. **Merge Bottleneck**: The final merge phase, which involves progressive communication and in-place merging in a tree structure, grows in cost as the number of levels increases. This overhead offsets the gain from reducing local sort time.

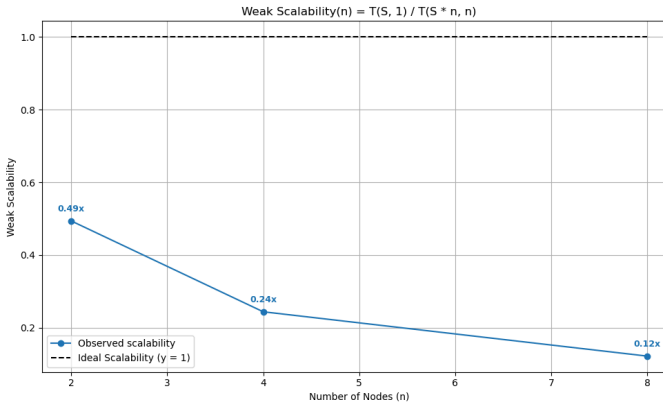
Nonetheless, it is important to observe that the **combined time of the base case and merge phases actually decreases** as the number of nodes increases. This suggests that the core parallelization logic scales only modestly, and the algorithm gains very little benefit from distributing both the sorting and merging work across multiple processes.

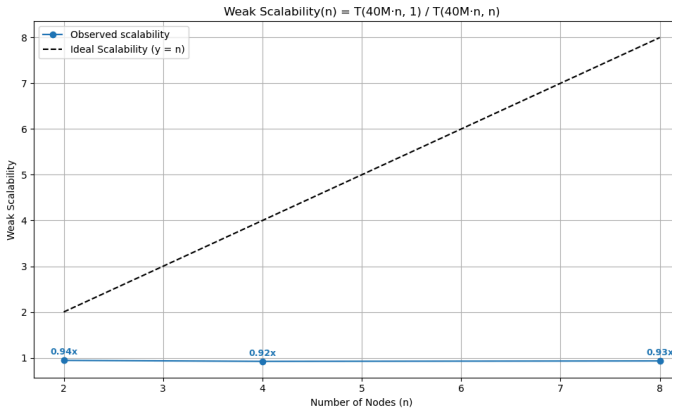
This analysis shows that The main limitation in scalability is therefore not due to computational imbalance, but rather to auxiliary communication costs, such as the initial data distribution via `MPI_Scatterv`, and the inherently sequential nature of the final steps of the algorithm.

Weak scalability

To evaluate weak scalability, I measured the total execution time while increasing both the number of nodes and the dataset size proportionally. Specifically, the dataset size was scaled linearly with the number of MPI processes, starting with 40 million records on a single node, up to 320 million records on 8 nodes. In other words, each configuration was tested using a global array size equal to 40M × *n*, where *n* is the number of nodes. The weak scalability is defined as: `Weak_Scalability(n)=T(S*n,1)/T(S*n,n)`. Conceptually, if we want the workload to scale perfectly, we expect that $T(S,\ 1) \approx T(S \cdot n,\ n)$. Therefore, the optimal value of the ratio above is n, meaning that the curve of ideal weak scalability grows linearly with the number of nodes (i.e., y=n). Alternatively, to reduce the number of execution with n=1, weak scalability is sometimes measured as: `Weak_Scalability(n)=T(S,1)/T(S*n,n)`. In this case, the ideal value is 1, and the scalability curve should be flat (i.e., y = 1) in the ideal case. In both formulations, the algorithm exhibits **poor weak**

scalability. The observed curves deviate significantly from the ideal trends, confirming that the parallelization does not scale efficiently as the number of nodes increases.





Also in this case, we can clearly identify the performance bottlenecks introduced by parallelization. The base case sorting time remains nearly constant across all configurations, despite the increasing dataset size. This confirms that the workload is evenly distributed across processes and that the local computation scales well as both the dataset size and the number of processors increase proportionally.

However, both the scatter and merge phases show a significant growth in execution time as the number of nodes and the overall dataset size increase. This is a direct consequence of the communication and coordination overhead introduced by parallelization, particularly due to the MPI-based data distribution and the hierarchical merging process. These phases become the dominant contributors to total runtime in distributed configurations, thus limiting weak scalability. In addition, it's important to consider that, as the merging progresses through successive levels, the number of active participants naturally decreases. At each level, only half of the processes are involved, until a single node remains responsible for performing the final merge. This last step is inherently sequential and centralized, regardless of the total number of nodes initially used. It represents a structurally non-parallelizable phase, which inherently limits scalability in distributed configurations.

As a result, the merge phase at higher levels become dominant contributors to the total runtime, effectively constraining the weak scalability of the system.

Nodes	Dataset size	Scatter Time	Base Case Time	Merge Time	Base + Merge	Total Time
1	40M	~0.38 s	~1.35 s	0.00 s	~1.35 s	~1.74 s
2	80M	~1.10 s	~1.34 s	~1.11 s	~2.45 s	~3.55 s
4	160M	~2.55 s	~1.36 s	~3.26 s	~4.62 s	~7.17 s
8	320M	~5.46 s	~1.33 s	~7.61 s	~8.94 s	~14.40 s