Report Assignment4 - Parallel MergeSort Implementation

Angelo Nardone

1 Introduction

In this project, one of the most well-known and extensively studied sorting algorithms, **MergeSort**, is considered. Specifically, two implementations of the algorithm are developed: a classical sequential version and a parallel version featuring two levels of parallelism: **intra-node parallelism** and **inter-node distributed parallelism**.

Intra-node parallelism is managed using **FastFlow**, a `C++` parallel programming framework designed for shared-memory systems. For inter-node parallelism, **MPI (Message Passing Interface)** is employed to enable distributed computation across multiple nodes. The study of the parallel implementation is divided into two parts: the first focuses solely on intra-node parallelism, while the second presents a **hybrid implementation**, combining intra-node and inter-node techniques.

At the conclusion of the implementation phase, an analysis of the results is conducted to determine whether the observed performance aligns with expectations. The evaluation is based on key performance metrics such as **speedup** and **efficiency**, along with an investigation of **strong** and **weak scalability curves**.

The remainder of the report is structured as follows. Section 2 provides a detailed description of the implementation phases and techniques, along with justifications for the design choices made. Section 3 presents a comprehensive performance analysis, structured into two comparisons: sequential vs. parallel execution on a single node, and sequential vs. distributed parallel execution. Finally, Section 4 concludes the report by summarizing the results, outlining the main development challenges encountered, and suggesting possible directions for future improvements.

2 Code Implementation

As mentioned in the introduction, this section describes the **implementation** of the different versions of MergeSort. Before presenting the code, however, it is necessary to introduce some important preliminary information. Specifically, two aspects will be addressed: the **type of data to be sorted** using MergeSort, and the **expected command-line** input format.

The first point concerns the data type to be sorted. As specified in the project guidelines, the objective is to sort a `struct` named `Record`, composed of two elements. The first element is an `unsigned long`, which serves as the key for sorting. The second is a `char[RPAYLOAD]` field, which is used to analyze the overhead introduced by varying `RPAYLOAD`. The structure of the `Record` is defined as follows.

Definition 1 (Struct Record)**.** Throughout this report, the term `Record` refers to the following data structure:

```
struct Record {
    unsigned long key;
    char* payload;
}
```

This structure includes a **constructor** for the `payload` field and implements the **Rule of Five** to ensure proper resource management. The `key` field is initialized using a dedicated function.

The second point relates to the command-line arguments required to run the program. In general, the code is executed using the following command:

```
./mergesort_implementation -s SIZE -r RPAYLOAD -t N_THREAD
```

where:

- `-s SIZE`: Specifies the **length of the** `Record` array to be instantiated. To improve readability, the code includes a function that allows the use of **abbreviated alphanumeric notations**, such as `K = 1,000` and `M = 1,000,000`. For instance, the command-line argument `SIZE=10K` will be interpreted as `SIZE=10,000`. If omitted, the **default value** is `SIZE=1,000,000`.

- `-r RPAYLOAD`: Defines the **size in bytes of the `payload`** field in the `Record` struct. The use of a **dynamic `char*`** rather than a statically sized array allows the payload size to be set via the command line. The **default value** is `RPAYLOAD=32`.
- `-t N_THREAD`: Indicates the **number of cores** to be used on a single node, applicable to both the parallel and distributed implementations. This flag is **not permitted in the sequential version**. The **default value** is `N_THREADS=8`.

Observation 1. It should be noted that the **number of nodes used in the distributed version** is also specified via command-line arguments. The command to launch the distributed implementation is as follows:

```
srun --mpi=pmix -N N_NODES -n N_NODES ./distributed_mergesort -s SIZE -r RPAYLOAD -t N_THREAD
```

Here, the `-N` flag specifies the **number of nodes** to be allocated, while the `-n` flag sets the **number of processes**. In this implementation, the number of processes always matches the number of nodes. Additionally, in order to execute the distributed version correctly, the code must be run from within a **compute internal node** using `srun`.

Before describing the implementations, it is important to highlight that all three versions of the code make use of a set of shared `.hpp` files containing **general-purpose functions** and **definitions** essential for their execution. Specifically, the following files are used:

- `config.hpp`: This file contains the **definitions of global variables** shared by all three implementations, such as `SIZE`, `RPAYLOAD`, and `N_THREADS`, along with their default values. The **definition of the `Record` struct** is also located here.
- `cmdline.hpp`: This file includes functions related to **command-line argument handling**. In particular, it contains the `parseCommandLine` function, which reads and validates command-line input, assigning values to the global variables defined in `config.hpp` accordingly. It also provides a `usage` function that prints the correct syntax for running the program in case of incorrect input.
- `general_helpers.hpp`: This file provides **utility functions** used throughout all implementations. Among these are `rand_init_ints`, which generates uniformly random keys for the `Record` array to be sorted, and `is_sorted_keys`, which checks the correctness of the sorting procedure. In addition, it includes `isNumber` and `parseLen`, both of which are used in `cmdline.hpp` to validate input and support abbreviated alphanumeric notations for `SIZE`.

The implementation of the algorithms for the different versions of the code is now discussed in detail.

2.1 Sequential Code: `mergesort_seq`

The description begins with the **sequential version**. Given that mergesort is one of the most widely known and studied algorithms, only a few essential clarifications are necessary.

First, the array of `Record` objects to be sorted is represented as a `std::vector<Record>` from the C++ Standard Library. This choice ensures that when sorting is performed based on the `key` field, the entire `Record` struct, including the `payload`, is **moved accordingly**. Moreover, `std::vector` automatically **handles memory deallocation** when it goes out of scope, making it convenient and safe to use. This rationale applies to all implementations, not just the sequential one.

Second, although the purpose of the sequential code is to serve as an efficient baseline, a custom implementation of mergesort was preferred over using a highly optimized library function. This decision was made to ensure a **fair comparison** with the parallel versions, as the internal behavior of library implementations may differ and introduce inconsistencies in the evaluation.

The algorithm itself is straightforward. Two `std::vector<Record>` instances are declared: `data`, the array to be sorted, and `temp`, a temporary buffer used during the merge process. The `key` values in `data` are initialized with uniformly distributed random values. These vectors are then passed to the function `mergesort_seq`, which checks whether the size of the subarray is below a **predefined threshold** (`BASE_CASE_SIZE`, set to 1000 in `config.hpp`). If so, it delegates sorting to the `stable_sort` library function. Otherwise, `mergesort_seq` recursively splits the array into two halves. After recursion, the sorted segments are merged using the `merge` function, which is not described here as its behavior is assumed to be well understood. Upon completion, the function `is_sorted_keys` is called to verify the correctness of the sort. Since `std::vector` is used, the order of the `payload` fields remains consistent with the order of the keys.

Observation 2. The function `stable_sort` was chosen over `sort` because **stability** is a desirable property of mergesort and is not guaranteed by other standard algorithms.

2.2 Parallel Code: `mergesort_par`

We now turn to the implementation of code with **intra-node parallelism**. As previously mentioned, this part of the work was developed using the `FastFlow` library, specifically relying only on its **building blocks**, without employing higher-level abstractions. For instance, in our case (implementing `MergeSort`) the most natural and efficient choice would have been to use FastFlow's built-in **divide-and-conquer** structure. However, for educational purposes, the primary objective was to understand the underlying principles and mechanisms of FastFlow. Such insights would not have been evident had we used a pre-packaged algorithmic skeleton. It is also important to remember that the number of FastFlow nodes (i.e., **threads**) is provided via the command line. Let us now begin the detailed code description.

The chosen structure for this implementation is a **farm**. This farm exhibits two key properties: it has **no collector node** (`farm.remove_collector()`), and its workers are allowed to **send information back** to the emitter (`farm.wrap_around()`). The farm is then constructed in the standard way: it includes an emitter node (implemented as a `Master` node in the code) and a set of worker nodes, each corresponding to a `Worker` instance. An initial check ensures that at least two threads are specified via `N_THREAD`. If not, the **sequential version** of the algorithm is executed instead. This is because with only one thread, a farm would consist solely of the emitter node, essentially reproducing the sequential algorithm but with additional overhead.

Then, also a custom `struct Task` is defined to encapsulate the information exchanged among nodes. Its definition is as follows:

```
struct Task {
    size_t left;       // left pointer
    size_t mid;        // middle pointer
    size_t right;      // right pointer
    bool is_sorted;    // flag to indicate sorting status
    int id;             // node identifier
};
```

At this point, the implementation can be described in two main phases: (1) the **partitioning and local sorting** phase, and (2) the **global merge** phase. We begin with the first.

In the **first phase**, after allocating the two vectors `data` and `tmp`, and initializing the `Record` keys (as in the sequential version), both vectors are sent to the `Master` node. In the Master's logic (specifically within the clause `if(task == nullptr)`), the input vector is divided into `N_THREAD` equal parts, and the start and end pointers of each part are computed. Then, the first `N_THREAD - 1` segments are assigned to worker nodes by sending a task using `ff_send_out(task)`. Each task specifies the left and right pointers of the segment to be sorted, the worker's ID, and sets `is_sorted = false`. The Master retains the last segment and processes it **sequentially**, returning `GO_ON`.

Each worker, upon receiving a task with `is_sorted = false`, performs a **sequential sort** on the segment, updates `is_sorted = true`, and returns the task to the Master. At the end of this phase, the Master will have `N_THREAD` sorted blocks. A visual representation of this phase is provided in Figure 5 shown in Appendix .

In the **second phase**, the Master coordinates the merge phase. This is facilitated through a utility function `send_task` and a `std::deque`. The deque keeps track of the start and end pointers of each sorted block, corresponding to the tasks from both the initial sorting phase and the subsequent merge phases. First, the Master **waits for all tasks** to be returned and deallocates their memory. Then, it iteratively traverses the deque **from left to right** (using `pop_front()`), pairing adjacent blocks. For each pair, a new task is created: `left` is set to the left pointer of the first block, `mid` to the left pointer of the second, and `right` to the right pointer of the second. The `is_sorted` flag is set to `true`. This new task is then dispatched to a Worker.

In this case, since `is_sorted = true`, the Worker performs a **sequential merge** of the two blocks and returns the result. The Master waits again for all tasks to return, deallocates their memory, and proceeds with another round of `send_task`, this time traversing the deque **from right to left** (via `pop_back()`). The rationale behind this alternation is detailed in Observation 3. In summary, the `send_task` function alternates between left-to-right and right-to-left deque traversal. This iterative process continues until only a single merge task is sent when using `send_task`. Once this final task is returned, the Master knows that the **full array has been merged** and is now sorted. At this point, an `EOS` (**End Of Stream**) task is broadcast to all Workers, signaling termination. Finally, a check is performed using `is_sorted_keys` to ensure that the output is indeed correctly sorted.

Observation 3. The reason behind alternating the deque traversal direction in `send_task` is as follows. When the initial value of `N_THREAD` is **odd** or **not a power of two**, it may happen that one block remains unmatched during a merge round. This unpaired block is generally smaller than the others. If the merging continues strictly from the left, this smaller block would remain at the end and be merged last, resulting in a **highly unbalanced merge** operation. By alternating the direction of traversal, this imbalance is mitigated, resulting in more evenly sized merges. An example of such balanced behavior is shown in Figure 6 shown in Appendix A. The choice of using a `deque` is clearly motivated by its ability to pop elements from both ends.

2.3 Distributed Code: `mergesort_mpi`

We conclude by describing the implementation of the code used for the **distributed version** of Mergesort. As previously mentioned, **MPI (Message Passing Interface)** was employed for this purpose, enabling data communication between nodes. The general idea behind this implementation is closely aligned with the one adopted in the parallel version using FastFlow. Moreover, FastFlow was reused within each single node to take advantage of **intra-node parallelism**. The fundamental steps of the implementation are now detailed.

Observation 4. It should be noted that the distributed version presented here is likely **not the most optimized** one. Due to time constraints, the first working version was retained. However, possible improvements and optimizations are discussed in the Section 4.1.

The first phase, data distribution, is handled similarly to the FastFlow version. A **farm-like structure** is implemented: the `Master` node, where the data is initially **allocated and initialized**, sends chunks of the input array to the `N_NODES`-1 worker nodes, dividing it into **equal parts** to ensure a balanced distribution. Communication is handled through standard `MPI_Send` and `MPI_Receive` calls.

To avoid **memory management issues** (especially when the input vector exceeds a certain threshold or the payload size exceeds `RPAYLOAD`) data is sent in chunks. During inter-node communication, three key pieces of information are transmitted: (i) the **number of records** being sent, (ii) all the **keys** of the records stored contiguously in memory, and (iii) all the **payloads**, also stored contiguously. Each node (including the master) then proceeds to sort its assigned block locally, leveraging the full number of threads specified by `N_THREAD`, using the FastFlow-based parallel sorting code previously discussed.

Observation 5. Memory-related issues stem from the fact that, in order to transmit record payloads, pointers cannot be used. Instead, the payloads must be stored in **contiguous memory** regions. When dealing with more than 50 million records, each with a payload size of 256 bytes, the allocation of such large contiguous memory blocks may fail due to fragmentation.

The second phase differs slightly from the FastFlow version. A dedicated function, `get_merge_partners`, is used by each node to precompute the nodes from which data must be received or to which it must be sent during the merge phase. As in the first phase, data is exchanged in chunks beyond a certain size. The merge phase follows a **reverse-tree structure**, allowing the merge to be completed in $\log$ `N_NODES` steps. Nodes responsible for sending their sorted data terminate their execution with an `MPI_FINALIZE` call. Nodes that receive data first perform a **sequential merge** with the locally available data and then recursively check whether they must send or receive more data. The final step always concludes with a node sending its data to the master node, which performs the final merge and validates that the entire array is sorted correctly.

3 Experiments and Results

This section presents an analysis of the behavior of the different implementations discussed previously, using a variety of **performance metrics**. Specifically, the behavior of the parallel version on a single node and that of the distributed version will be studied separately. For the FastFlow-based parallel implementation, the analysis will focus on **speedup** and **efficiency**, compared to the sequential version. In contrast, the analysis of the distributed implementation will focus on **weak** and **strong scalability**. Assuming these metrics are already well known, the experimental setup will be outlined before presenting the results.

To evaluate the behavior of each implementation, the programs were executed while systematically varying the parameters `SIZE`, `RPAYLOAD`, `N_THREADS`, and `N_NODES`, depending on the implementation being tested. Each configuration was executed three times on **internal cluster nodes**, and the **average execution** time across all runs was used as the reference value. The exact parameters used for running the programs are reported in Table 1.

Implementation	`SIZE`	`RPAYLOAD`	`N_THREADS`	`N_NODES`
Sequential	1M, 10M, 100M	8, 16, 32, 64, 128, 256	-	-
Parallel	1M, 10M, 100M	8, 16, 32, 64, 128, 256	1, 2, 4, 8, 16, 32	-
Distributed	1M, 2M, 4M, 8M, 10M, 100M	8, 16, 32, 64, 128, 256	8, 16, 32	1, 2, 4, 8

Table 1: Comparison between different implementations based on various parameters.

3.1 Parallel Version

The analysis begins with the parallel version. As previously mentioned, this version is evaluated using the metrics of **speedup** and **efficiency**. Figure 1 shows the speedup of the parallel version compared to the sequential one when `SIZE` is fixed at 100M. The **different curves** represent the results obtained with **varying payload sizes**.

As observed in the figure, the speedup achieved is **not particularly high**: the maximum value reached is approximately 3.5. This result is largely attributable to the nature of the mergesort algorithm, which, due to its limited computational intensity, is not inherently well-suited to parallelization. Specifically, mergesort is a **memory-bound** algorithm rather than compute-bound. The most time-consuming operation is the merging phase, which involves frequent memory accesses and data movement, rather than arithmetic operations. These characteristics limit the benefits typically gained from multi-threading, especially in environments where memory access becomes a bottleneck.

From the figure, it is also evident that the best results are obtained with 16 threads, while a degradation in performance occurs when using 32 threads. This is likely caused by the increased overhead in coordinating communication among FastFlow nodes, particularly during the **merge phase**. With a higher number of threads, the cost of synchronization and task scheduling tends to grow, often outweighing the gains from parallel execution. It is also worth noting that the expected trend (where a smaller payload leads to faster execution) is not always observed. This can be explained by the current implementation, which handles payloads through **pointers**. As a result, the actual memory overhead may not vary significantly with different payload sizes, and the access patterns may introduce cache misses or memory contention that affect performance in less predictable ways.

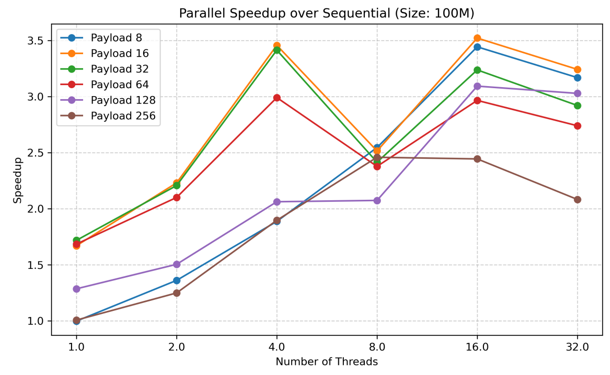

Figure 1: Speedup of the parallel version compared to the sequential baseline, with `SIZE` fixed at 100M.

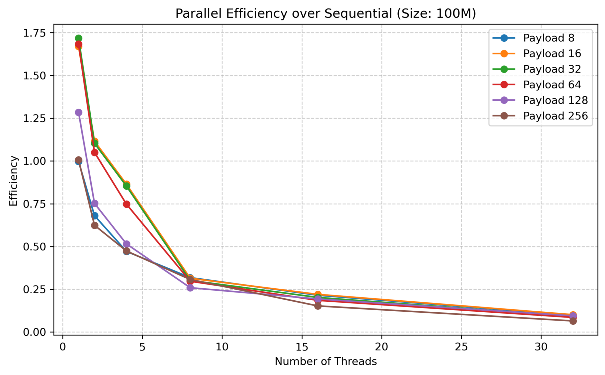

Figure 2: Efficiency of the parallel version with `SIZE` fixed at 100M.

Figure 2 shows the efficiency for the same experiments. As illustrated in the graph, the trend is similar across all payload configurations. The efficiency **decreases in a roughly logarithmic** fashion as the number of threads increases. This behavior highlights that although some speedup is achieved, it remains limited. The overhead associated with managing multiple threads (such as task distribution, synchronization, and memory access contention) grows faster than the computational benefits, leading to lower efficiency. This result reinforces the conclusion that mergesort, due to its nature, does not scale well when parallelized beyond a certain threshold.

3.2 Distributed Version

The analysis now turns to the distributed implementation. Figure 3 presents the results of **strong scalability**, with `SIZE` fixed at 10M. As shown, the execution time using only **one node** (corresponding to the parallel version running on a single node with FastFlow) is comparable to the execution time using eight nodes. This highlights the significant impact of distribution **overhead**, which tends to overshadow the potential gains of parallel execution across nodes.

Moreover, it can be observed that, in this case, **execution time increases with the payload size**, due to the growing amount of memory that needs to be transferred between nodes. In particular, the curve corresponding to `RPAYLOAD` 256 shows significantly lower execution times compared to the others. This behavior can be explained by the activation of the chunking mechanism for this payload size, which fragments the data into smaller pieces during transmission. While this avoids memory overflow issues, it further increases communication overhead. This becomes even more evident by observing Figure 8 in the appendix, where the same experiment is repeated with `SIZE` set to 100M. In this case, the chunking mechanism is always triggered due to the larger data size, and as a result, **all curves appear similar** and exhibit significantly lower execution times across all payload sizes.

Figure 4 displays the results of the **weak scalability** test. In this experiment, the base case starts with a size of 1M on a single node, and the size is doubled for each doubling of the number of nodes. The plot clearly shows a

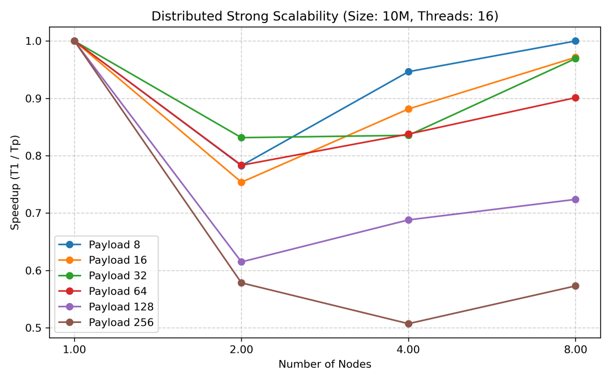

Figure 3: Strong scalability of the distributed implementation, with `SIZE` fixed at 10M and `N_THREADS` set to 16.

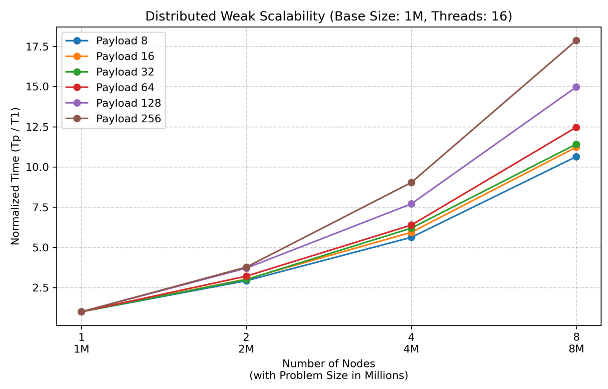

Figure 4: Weak scalability of the distributed implementation, with base `SIZE` set to 1M and `N_THREADS` set to 16.

rise in execution time as the number of nodes increases. While for smaller payloads the increase may appear nearly linear, for larger payloads the curve tends to **grow almost exponentially**. This once again emphasizes how the communication overhead dominates the computational gain when more nodes are added. The results suggest that an **alternative strategy** (such as minimizing the amount of data exchanged between nodes) could significantly improve performance.

4 Conclusion

This work presented and analyzed three different implementations of Mergesort: sequential, parallel, and distributed. Each implementation was discussed in terms of its structure and execution phases, along with the obtained performance results. It was observed that the parallel version running on a single node achieved reasonably good results—especially considering the inherently limited parallelism of the problem. On the other hand, the distributed version did not yield equally satisfactory outcomes. Nevertheless, the overall result is considered positive, particularly because the main objective of the assignment was to become familiar with distributed and parallel programming using **MPI** and **FastFlow**, respectively. This goal has certainly been met, and the practical understanding of both libraries has significantly improved throughout the development of this work.

4.1 Main Difficulties and Future Improvements

As anticipated, one of the major challenges was the limited amount of time available. In particular, the necessity to complete the assignment by May 27th (due to personal reasons) resulted in fewer days for final refinements and optimizations. Moreover, entering the worlds of MPI and FastFlow was initially **non-trivial**. However, thanks to the lecture slides and the code shared by the professor via Teams, it was possible to grasp and manage the basic concepts with reasonable ease. The most complex aspect was undoubtedly the management of **MPI communications** when handling large amounts of data. This issue caused various failures before implementing the chunk-based transmission phase.

Regarding possible improvements, several ideas have been identified. For the FastFlow implementation (which is the more satisfactory of the two) two primary enhancements could be considered: (i) avoiding repeated allocation and deallocation of tasks by maintaining a **deque of reusable tasks**, thus deallocating only those that are strictly necessary; (ii) exploring a version in which the merge phase is also performed through FastFlow nodes, potentially using a peer-to-peer communication model among workers, excluding two nodes from the worker pool and adopting an **all-to-all strategy**.

As for the distributed version, several improvements appear both possible and necessary. A first step would be to test the usage of `MPI_Isend` and `MPI_Irecv` instead of blocking `Send` and `Receive` operations, combined with a proper use of `MPI_Wait`. These non-blocking calls were in fact used initially; however, during the debugging phase, a switch to blocking operations was made to simplify error identification, and due to time constraints, it was not possible to revert back to the original approach.

A potentially more impactful improvement would be to **avoid sending the payloads** of records between nodes altogether. Instead, only the record keys could be sorted and transmitted, along with a data structure to track the correct ordering. The payloads could then be locally rearranged at the end of the process based on this ordering. This approach would significantly **reduce the communication overhead** and could be efficiently implemented using `MPI_Scatterv` and `MPI_Gatherv`.

A Appendix - More Figures

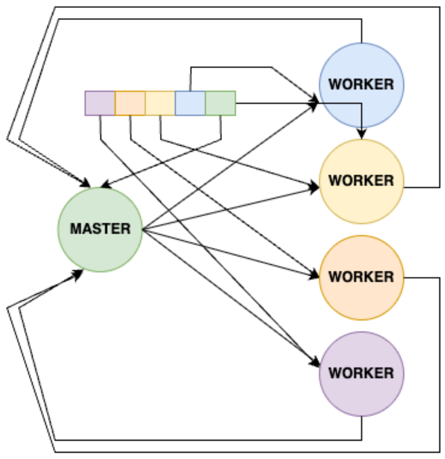

Figure 5: Illustration of Phase 1 of the parallel mergesort: initial data splitting and local sorting.

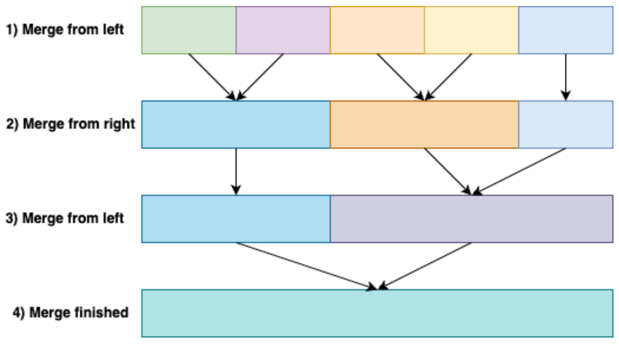

Figure 6: Example of the merging process during Phase 2 of the parallel mergesort.

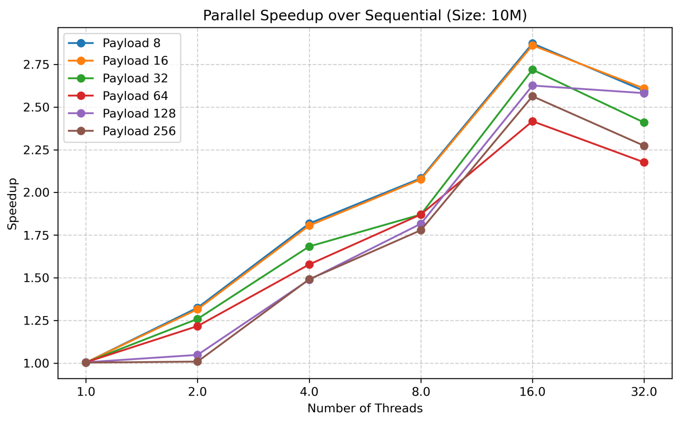

Figure 7: Speedup of the parallel version compared to the sequential baseline, with `SIZE` fixed at 10M.

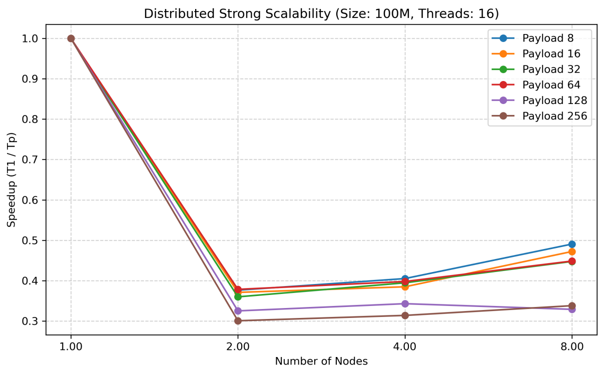

Figure 8: Strong scaling of the distributed version, with `SIZE` fixed at 100M.