Project - MergeSort

Francesco Fiaschi

September 2025

1 General Implementation Details

The first part of the implementation presents the data structures used in the program. The main structure is `Record`, which represents a single record with a key, the length of the payload, and a flexible array for the payload with a dimension between `[8, PAYLOAD_MAX]`.

The other structure is `RecordHeader`, which is used to store the `key` for sorting the array and the `original_index` about each record. This approach allows records to be sorted only by the key, while also enabling data exchange between processor with minimal data movement. The `original_index` is crucial for maintaining the relationship between the sorted records and their original positions in the input array.

At the end of the sorting of each chunk in every rank, the worker sends the sorted chunk to the main processor (Rank 0) with an array containing only the `original_index` of each record in the chunk. This structure is defined in Listing 1.

The general structure of the code is as follows:

1. Create the file with random records.
2. Load the records from the file into an array of `Record` structures.
3. Divide the array into smaller chunks for parallel processing.
4. Sort each chunk in parallel.
5. Merge the sorted chunks back together to form the final sorted array.

Listing 1: Struct RecordHeader

```
struct RecordHeader {
    unsigned long key;
    size_t original_index;
};
```

2 Single Node Version

To compare the parallel version we report the perform metrics of the sequential version. This is implement with `std::sort` which is very efficient and cache friendly because it operate only on pointers without moving the payload.

SIZE	PAYLOAD	Sorting Time (s)
100000	128	0.0097
	256	0.0115
	512	0.0135
1M	128	0.17
	256	0.195
	512	0.283
10M	128	2.7
	256	3.106
	512	3.865

Table 1: Time for sorting the array of record

2.1 FastFlow with Farm

Write Record to File

The first step to create the file, with a number of random records selected from the command line argument, is implemented with `farm`, which enables the use of parallelization by dividing the number of requested records into equal chunks.

The farm is composed of the standard `Emitter`, which assigns the tasks with Round Robin scheduling. The `Worker` and `Collector` are explicitly declared. The worker receives the number of records to create and uses the function `createRandomRecord` to generate the records.

Each worker creates chunk records, but it doesn't write file because of the concurrency problem. The collector gathers all the chunks records and iteratively writes them to the file in sequential mode, without any concurrency problems.

Load from File

The load part, however, is more complex to implement in parallel version. We need to read the records from the file in parallel, but we don't know how to split the file because the records have variable payload sizes. Due to these issues, and also because reading sequentially before parallelizing the payload is useless, this part is implemented in sequential mode, reading all the records from the file and storing them in an array of `Record` structures.

Sorting Record Array

This part is implemented in parallel using the FastFlow farm. The farm is composed of specific implementation of emitter, worker and collector components.

The emitter is responsible for calculating and sending the appropriate tasks to the workers. It also handles the last chunk of records (which may be smaller than `chunk_size`) by using `std::min(i + chunk_size, arr.size())` to avoid out-of-bound access.

The sorting of the record array is performed using a parallel merge sort algorithm. The array is divided into smaller chunks, trying to balance the workload among the threads.

Every worker thread sorts its chunk of the array independently with the standard `sort` function that is very efficient for medium to small arrays.

The collector, once it receives any sorted chunks, performs the final merge of all sorted chunks into a single sorted array. This merging process is also done in incremental mode to improve performance: every time a chunk of records is sorted, it is merged with the already sorted array.

To validate the correctness of the sorting, we control the order at the end of the sorting process, ensuring that each record's key is less or equal than the next record's key. The comparison of execution time with the sequential version is shown in the Results section.

2.1.1 Results

In Table 2, there are the result of various test with different array size, payload and number of thread. In order to test the speed and the overhead generated from this optimization with FastFLow Farm.

The data in table underline that with all array size and all payload, the execution with one thread is useless and required a lot of time. On the other hand use a large number of thread generate a lot of overhead, which can lead to diminishing returns in performance, as we can see in the table for each value of time with 8 threads. The best trade-off is obtained when using 4 threads, that increases the performance but without the cost of overhead. The speedup is very significant, reaching up to 4.4x with 4 threads for 10M records with a payload of 512 bytes. As we can see from the table, the speedup is more pronounced with larger payloads and array sizes, indicating that the parallelization benefits are more significant when there is more data to process. We notice that the speedup is not linear with the number of threads, which is expected due to the overhead of managing multiple threads and the limitations of the hardware, in fact with 8 threads the speedup is lower than 4 threads in several cases.

In the same way, the efficiency decreases with the increase of the number of threads without a gain in performance over 4 threads.

Strong scalability

The strong scalability of the proposed parallel sorting algorithm is evaluated by measuring the execution time as the number of threads increases, while the problem size remains constant.

From the Graph 1, we can observe the performance trends as we scale the number of threads. The graph show which is the point where the overhead starts to dominate the execution time, in fact with 8 thread the performance drops significantly and the speedup is lower than 4 threads. But the program have a good scalability up to 4 threads, beyond which the performance degrades due to the overhead of synchronization and memory limits.

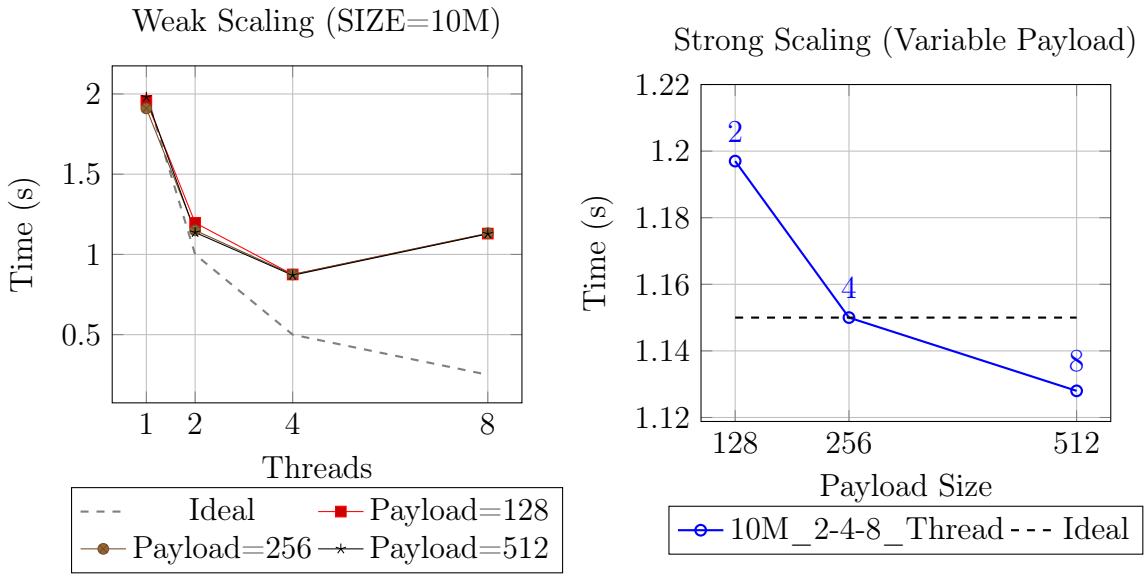
SIZE	PAYLOAD	Threads	Time (s)	Speedup	Efficiency
100K	128	1	0.008	1.51	1.51
		2	0.005	2.44	1.22
		4	0.005	2.87	0.71
		8	0.005	2.12	0.265
100K	256	1	0.008	2.2	2.2
		2	0.005	2.3	1.15
		4	0.005	2.3	0.575
		8	0.006	1.91	0.239
100K	512	1	0.006	2.25	2.25
		2	0.005	2.7	1.35
		4	0.005	2.7	0.675
		8	0.005	2.7	0.337
1M	128	1	0.129	1.31	1.31
		2	0.080	1.59	0.795
		4	0.07	1.79	0.447
		8	0.09	1.40	0.175
1M	256	1	0.127	1.53	1.53
		2	0.080	2.43	1.215
		4	0.068	2.86	0.715
		8	0.092	2.12	0.265
1M	512	1	0.129	2.2	2.2
		2	0.08	3.53	1.765
		4	0.069	4.1	1.02
		8	0.09	3.14	0.392
10M	128	1	1.957	1.38	1.38
		2	1.197	2.25	1.125
		4	0.875	3.08	0.77
		8	1.13	2.39	0.3
10M	256	1	1.91	1.62	1.62
		2	1.15	2.7	1.35
		4	0.875	3.55	0.89
		8	1.132	2.74	0.34
10M	512	1	1.98	1.95	1.95
		2	1.137	3.4	1.7
		4	0.870	4.4	1.1
		8	1.128	3.42	0.43

Table 2: Execution time of FastFlow experiment

Weak scalability

The weak scalability is assessed by increasing the problem size proportionally with the number of threads. The execution time remains relatively constant between 1.1 and 1.2 seconds, demonstrating that the algorithm can handle larger datasets without a significant increase in processing time.

In the Graph 1, we can observe that instead of have a linear trend, the addition of threads does lead to a decrease in execution time, caused a slight improvement in performance. This behavior indicates that the algorithm scales well with increasing problem sizes, but there are still some overheads that prevent it from achieving perfect weak scalability especially with 8 threads.



Graph 1: Scaling of FastFlow

2.2 OpenMP

In OpenMP, the implementation follows a similar approach to the FastFlow version, but leverages OpenMP's built-in parallelization capabilities.

Write and Load File

The writing part is implemented in parallel using OpenMP's `#pragma omp parallel for` directive, which allows multiple threads to create records concurrently. Each thread generates a chunk of records and writes them to the file within a critical section to avoid concurrency issues.

The number of threads is determined by the pragma directive, which allocates and uses the available CPU cores, if it was not passed from the command line.

The load part is implemented in sequential mode, similar to the FastFlow version, reading all records from the file and storing them in an array of `Record` structures, for the same reasons explained in the FastFlow section.

Sorting Record Array

To sort the record array instead of dividing it into chunks, we can use a custom version of merge sort. The structure is the same as the normal mergesort but the recursive calls are parallelized using OpenMP tasks. The main call is made in a parallel region with a single thread, which creates tasks for each recursive call. The task `#pragma omp task shared(arr, temp)` creates a new task for each recursive call, allowing them to run in parallel but with a shared context. This approach allows for better load balancing and can lead to improved performance on multi-core systems. The threads are created dynamically as needed, allowing for efficient use of resources. The maximum number of threads is used when the recursive calls

saturate the available hardware threads. In this situation, we achieve maximum parallelism, with both its advantages and disadvantages.

2.2.1 Results

The result in Table 3 shows the performance of the OpenMP implementation. The time execution increase a little bit with the payload size. The speedup, especially with small array size, is very bad. With array size 10M the speedup reach also 1.5x but only with 8 thread.

The OpenMP implementation shows a more consistent improvement in performance as the number of threads increases, especially for larger payloads and array size. But the speedup show the limitations of this implementation especially with small array sizes, where the speedup is not significant and is lower than the sequential version.

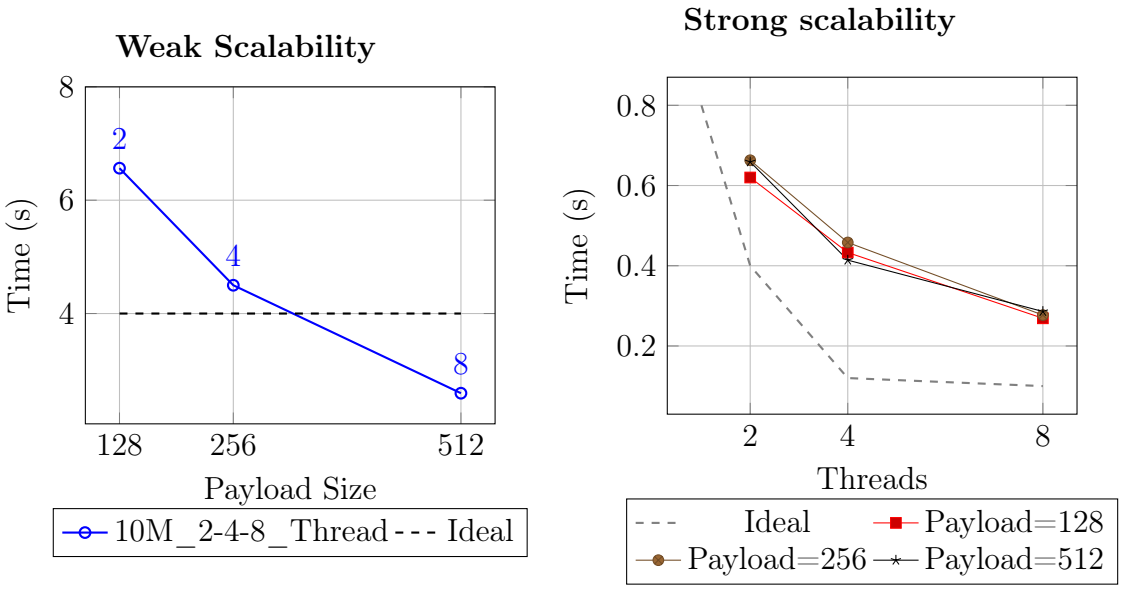
SIZE	PAYLOAD	Threads	Time (s)	Speedup
100k	128	2	0.0572	0.17
		4	0.0411	0.17
		8	0.0221	0.236
100k	256	2	0.0593	0.194
		4	0.0418	0.275
		8	0.0230	0.5
100k	512	2	0.0626	0.21
		4	0.0419	0.32
		8	0.0253	0.53
1M	128	2	0.620	0.274
		4	0.433	0.39
		8	0.269	0.631
1M	256	2	0.663	0.29
		4	0.458	0.42
		8	0.277	0.7
1M	512	2	0.659	0.429
		4	0.414	0.68
		8	0.286	0.99
10M	128	2	6.565	0.41
		4	4.385	0.62
		8	2.443	1.11
10M	256	2	6.771	0.46
		4	4.499	0.7
		8	2.814	1.10
10M	512	2	7.071	0.55
		4	4.756	0.81
		8	2.594	1.5

Table 3: Times, speedup and efficiency of OpenMP

Scalability

In the graph of weak scalability, we can observe that the trend tends to go down and the difference between with 128 and 256 of payload is large. This indicates that the model do not scale well, even if the performance improves with larger payloads.

On the other hand, the strong scalability graph shows a slight improvement, especially in the step from 4 to 8 threads, with slight good improvement up to 4 threads. However, the trend is not sufficient to indicate a clearly positive behavior.



3 Multi Node Version

In a multi node version, the implementation uses MPI (Message Passing Interface) to distribute the workload across multiple processes. MPI throws a number of processors equal to ranks specified in the command line when the program is executed. To start the parallel section we need to call `MPI_Init(&argc, &argv);` which initializes MPI environment and prepares it for communication. From this moments everything runs in parallel.

The creation of records is performed in parallel across the nodes, each node handling a portion of the total array. This method allows for better scalability and utilization of resources in a distributed environment. All processes generate records concurrently but also use the `farm` to parallelize again the record creation, in the same way as in the FastFlow version.

Each MPI workers create `farmCreation`, tha farm is composed by the standard emitter, a specific implementation of worker which takes in input the number of record to generate and the maximum payload size. The farm then use a collector to take all buffer from the workers and write records concurrently with `MPI_File_write_ordered` to the file. The collector try to write all the records to the file in a single operation, minimizing the number of I/O calls and improving performance, but if the buffer size exceeds the maximum limit (`INT_MAX`), it will split the write operation into multiple calls.

The collector used `MPI_File_write_ordered`, which allows all processes to write to the file concurrently based on the order of calls. In this way, the write operations can be performed in a simple way but not in a fully parallelized manner.

During the development process, the first version of the code used an approach similar to the single node version: the writing of the records was done sequentially by the main process (Rank 0) to avoid concurrency problems through the use of

`MPI_Send` and `MPI_Recv` functions. The main process sent the number of records to create to each worker process, which creates its own chunk of records and sent them back to the main process for writing to the file. This approach allows for better load balancing and can improve performance on multi-core systems, but the main processor (Rank 0) become a bottleneck due to the handling both MPI communication and I/O operations.

To avoid this bottleneck, one possible solution is to implement a more distributed approach, where each worker process is responsible for writing its own records to the file. This can be achieved by using the MPI I/O functions, which allow concurrent file access by multiple processes.

We made also the MPI+OpenMP version that uses MPI I/O to enable parallel writing of records to the file. Each process writes its own chunk of records directly to the file, eliminating the bottleneck of a single process handling all I/O operations.

The main function used is `MPIFile_write_at_all`, which allows all processes to write to the file concurrently based on the predetermined offset. In this way, the write operations can be performed in parallel on different chunks, reducing the contention on the main process.

Due to the dynamic payload size, each processor needs to know the bytes that every other rank has written on the file. This can be achieved by using a collective function among all processes called `MPI_Allgather`. This function gathers the local size from each process and distributes it to all processes, enabling each one to calculate its own offset for writing to the file. The final array is identical on all processes, ensuring consistent access. This approach allow the fully parallelize and it works very well compare to use `MPI_File_write_ordered`, which is not fully parallelized.

The load part is implemented in sequential mode, similar to the FastFlow and OpenMP versions, reading all records from the file and storing them in an array of `Record` structures, for the same reasons explained in the FastFlow section.

The sorting part uses the MPI parallelism to split the workload among all processors. To distinguish the different processes, the `MPI_Comm_rank` function is used to obtain the rank of the current process. This allows us to identify each process and assign tasks properly.

The main process (Rank 0) is responsible for dividing the array into chunks and distributing them to the worker processes. To achieve the goal, we need to declared some global variables to store the total number of records in the file, because each worker process needs to know the total number of records to correctly calculate its own chunk size. However, the loading process is costly in term of both memory usage and execution time in fact to resolve this issues, the main process loads all records and then broadcasts to all worker only the number of records using `MPI_Bcast`.

In order to send the chunk of records to each worker process, we can use the `MPI_Scatterv` function. This function allows us to send variable chunks with different sizes of data to each process, which is useful when the array size is not perfectly divisible by the number of processes. In fact in this way, the reminder of the division

will be distributed among the first processes until the reminder is zero.

The first step is to send the chunk size of each process using the `MPI_Scatter` function in the parallel part of the program. Each process receives its chunk size and allocates memory for its local array of records. After this, each process have the exact number of `RecordHeader` that it will receive from the main process with the `MPI_Scatterv` function. The Scatterv function need that each rank have its own allocated buffer to correctly store the incoming data.

In Rank 0 we send the buffer containing the `RecordHeader` structures to each worker process using the `MPI_Scatterv` function. The reason for the difference position of scatter and scatterv is that the first one send only the chunk size, while the second one send the actual data with different size, that it is only present in the main processor which it made the load from file, otherwise, the function could be called out of the Rank 0.

On the other hand, each worker process receives its chunk of `RecordHeader` structures through the `MPI_Scatterv` function.

The RecordHeader structure is used to reduce the data passed between processes, it only contains the key and the original index of each record to recreate the original record later. This approach reduces the communication overhead and avoids fluctuating performances based on the payload size.

Each worker uses the received `RecordHeader` structures to order the RecordHeaders based on the key, this value correspond to the true data in the original array at position saved on the original index. Instead of using a sequential sort, each worker can use a farm of threads to sort its local chunk in parallel like in FastFlow single node. The farm is composed by the standard emitter, a worker which takes in input the local chunk of `RecordHeader` and sort it with the standard `sort` function and a collector to take all sorted buffer from the workers and merge them two by two when arrive.

When the sorting is completed, each worker process (Rank$\neq$0) sends an array of integer representing the sorted `original index` of its local chunk. These indexes indicate how to reorder the global record array according to the key order.

The main process posts non-blocking receives (`MPI_Irecv`) for each worker, in this way it can overlap communication with computation. In fact, while the workers are sorting and sending their chunks, the main process concurrently sorts its own local chunk, meanwhile waits to receive data from workers.

Once its local sort is completed, Rank 0 waits for all pending communications (`MPI_Waitall`) and then reconstructs the global sorted array by merging its own sorted chunk with the indexes received from the workers. This part is done using the standard `merge` function to ensure efficiency. The merge functions use as offset the array `sendcounts` create for the `Scatterv` function, which contains the number of records for each rank.

Finally, the main process writes the sorted records to the output file using MPI I/O functions. Each process writes its own chunk of records directly to the file,

eliminating the bottleneck of a single process handling all I/O operations.

3.1 Results

The results of the MPI+FF implementation are summarized in Table 5.

The results show that the implementation of MPI+FF is effective in reducing the overall execution time, especially for larger datasets. The use of FastFlow for the sorting phase allows for better utilization of available resources, leading to improved performance.

The reported execution time is the sum of the time spent in the header sending phase, the local and distributed sorting, and the final merge. The total time is significantly lower compared to the standard sort implementation reach a speedup of 3.87x.

We choose to use FastFlow instead of OpenMP for the parallelization of the sorting phase because it provides a fast execution environment. FastFlow's lightweight threads and efficient scheduling make it well-suited for the parallelism required in the sorting phase. Unlike OpenMP, which relies on compiler directives and may introduce additional synchronization costs. One other things to consider is that FastFlow in single mode version show best result compared to the OpenMP version, which obtains a lower speedup and a higher execution time, even the standard sort.

The implementation with OpenMP does not show any improvement. In fact, the execution time is much higher compared to the standard sort, probably due to an error in the MPI+OpenMP implementation.

The results in Table 4, show that the execution time remains essentially constant, around 10 seconds, regardless of the number of nodes, threads or dataset size. This suggests that a part of the code introduces an almost constant execution overhead, which acts as a lower bound for the total runtime. Even though the workload is distributed in parallel, there is a fixed cost that cannot be reduced by increasing computational resources.

As a result, the total time is not only determined by the distributed sorting phase but is also dominated by this serial/communication component, which represents the actual bottleneck of the system. Nevertheless, we can assume that the sorting and merging phase is not the bottleneck, since the time of this part is measured separately and turns out to be very low.

The OpenMP version remains more readable and maintainable, but the performance is not satisfactory, the FastFlow version shows better performance and scalability however maintainability and readability are still acceptable. The main advantage of the FastFlow version is its ability to efficiently utilize multiple cores and nodes, leading to significant speedups shown in the results table.

The OpenMP implementation of file writing achieves higher performance compared to the FastFlow version. In the FastFlow case, the collector used `MPI_File_write_ordered`, which allows all processes to write to the file concur-

rently but strictly following the order of the calls. This approach simplifies the implementation of the write operations, but it does not fully exploit parallelism. In contrast, OpenMP can leverage true parallelism through with MPI_IO, enabling multiple threads to perform concurrent writes. As a result, OpenMP achieves a more efficient utilization of resources and can significantly reduce the overall time spent on I/O operations.

SIZE	PAYLOAD	NODES	THREADS	Totale (s)
100K	512	2	4	10.436
		4	2	10.201
		8	2	10.739
1M	512	2	2	9.945
		4	2	10.324
		8	2	10.159
10M	512	2	2	10.171
		4	2	10.185
		8	2	10.230

Table 4: Sort time of MPI+OpenMP

The main advantage of using MPI I/O is that it enables true parallel file access, removing the bottleneck of a single process and improving scalability in distributed environments. However, MPI I/O also introduces additional complexity, as processes must correctly calculate offsets and coordinate access patterns to avoid data corruption and ensure file consistency. Moreover, offset calculation generates some overhead and cost in terms of time execution.

The results in Table 5 show the performance of the MPI+FastFlow implementation across different configurations. The execution time is measured for various combinations of array sizes, payload sizes, number of nodes, and number of threads. The execution time remains relatively constant across different configurations of payload sizes and always less than the sequential one. This indicates that the implementation is efficient in handling the workload distribution. The efficiency column shows that the FastFlow implementation achieves good resource utilization is some cases.

The speedup achieved with any number of nodes is very significant, but in some cases with a use of a number of nodes and threads both greater than 4, the performance improve less than the other configuration, especially with small array size. In fact, with big array sizes, the best configuration with 8 nodes between all number of thread is 8 threads, which reach a speedup of 3.55x with 512 bytes of payload. This value is still less that the configuration with fewer nodes and threads.

As we can see from the table, the speedup is more pronounced with larger payloads and array sizes, achieving up to 3.87x speedup with 2 nodes and 2 threads for 10M records with a payload of 512 bytes. This configuration is the best performing one in our tests, and also this is the most efficient in terms of resource utilization which obtain 0.967 of efficiency. The others obtain a optimal speedup but the number of nodes and threads is high and reduce the efficiency of the configuration.

SIZE	PAYLOAD	NODES	THREADS	Totale (s)	Speedup (x)	Efficiency
100K	128	2	2	0.008	1.20	0.3
		4	2	0.009	1.13	0.189
		8	2	0.009	1.03	0.247
100K	256	2	4	0.009	1.34	0.223
		4	2	0.009	1.33	0.221
		8	2	0.009	1.23	0.123
100K	512	2	4	0.009	1.56	0.389
		4	2	0.009	1.48	0.247
		8	2	0.01	1.38	0.138
1M	128	2	4	0.098	1.74	0.289
		4	2	0.111	1.53	0.255
		8	4	0.128	1.32	0.110
1M	256	2	2	0.098	1.99	0.332
		4	2	0.114	1.72	0.286
		8	2	0.134	1.45	0.091
1M	512	2	4	0.098	2.90	0.483
		4	2	0.11	2.58	0.430
		8	8	0.136	2.07	0.130
10M	128	2	2	1.019	2.65	0.663
		4	4	1.174	2.3	0.287
		8	8	1.302	2.07	0.130
10M	256	2	2	1.004	3.09	0.774
		4	4	1.136	2.73	0.342
		8	8	1.222	2.54	0.159
10M	512	2	2	0.999	3.87	0.967
		4	2	1.192	3.24	0.541
		8	8	1.214	3.18	0.199

Table 5: Sort time of MPI+FF

The trend observed in the results indicates that as the number of threads increases, the speedup not only increases but in many cases the best configuration is achieved with 2 nodes and 2 threads. This suggests that introduce a lot of threads and nodes do not improve significantly the performance, but only increase the overhead and reduce the efficiency.

4 Discussion

The first implementation of the multi node version split the work with MPI to generate the record, but the write remain sequential. Then to improve performance, we use the MPI_IO, which increases the speed of creation. However, the write operations become parallelized, allowing multiple threads to write to the file concurrently based on the order of calls.

To avoid the use of a lot of memory and to reduce the communication overhead, each worker process only receives the `RecordHeader` structures and send only a vector of integer containing the original index of each record. This approach minimizes the amount of data transferred between processes, leading to improved performance.

The selection of FastFlow over OpenMP for parallelization in the multi node version was driven by several factors. FastFlow provides a more efficient and flexible framework for implementing parallel algorithms. Additionally, performance tests

indicated that FastFlow allows better results in this specific context, making it a more suitable choice for the implementation.

Also, the MPI+FastFlow version demonstrated improved performance compared to the OpenMP implementation, particularly in terms of execution time and speedup which overcomes the standard sort. From the test the time of execution of Rank 0 is often lower then the other rank execution.

In the code there are some potential bottlenecks:

1. Communication overhead in the MPI implementation, especially for small array sizes, but this is mitigated by transferring only necessary data (RecordHeader and original indexes);
2. Write phase, where the use of MPI_IO, it's not fully parallel but it is better than a single process write;
3. Loading file in memory sequentially;
4. In FastFlow, the overhead of thread management and synchronization can impact performance particularly with small size array;

One potential bottleneck is the Rank 0 process, which is responsible for sorting its part of the array, receiving the sorted chunk and merging them. To mitigate this, we overlapped the communication and computation phases, in fact we use the `MPI_Irecv` non-blocking communication function to overlap with the computation of the local sort.

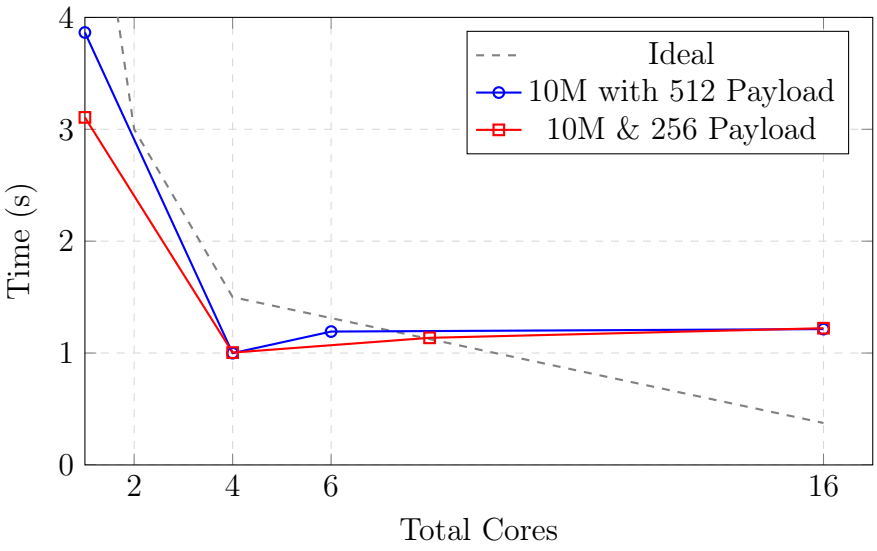
The cost model of the implementation can be summarized as follows:

- Communication cost for sending RecordHeaders and original indexes, as each process sends a small fixed-size message. This cost is mitigated by transferring only necessary data.
- The local sort phase using FastFlow, this part is the most time-consuming, but it is efficiently parallelized.
- The final merge phase, which is performed by the main process (Rank 0) and involves merging sorted chunks from all processes. This part can be a bottleneck if not managed properly, but in our case is very fast.

$$T(\text{Tot}) = T(\text{ArrayDim})T(\text{CommHeader})+T(\text{LocalSort})+T(\text{CommIndex})+T(\text{Merge})$$

In this function we have:

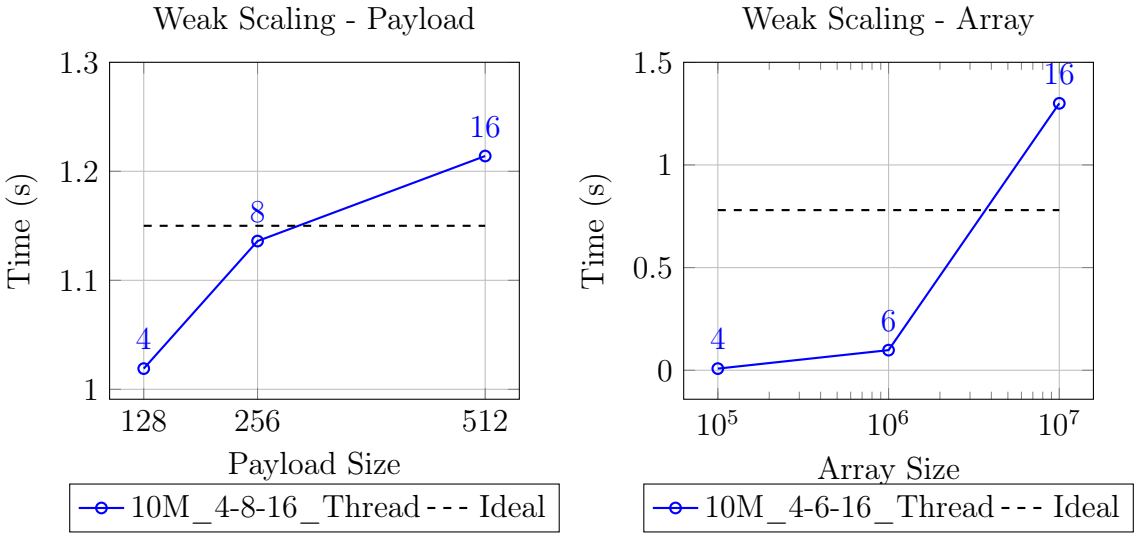
- $T(\text{ArrayDim}) = l \cdot \log p + \beta \cdot m$. This term represents the communication cost for sending the array dimension structures to each process, the latency is calculated for each send but we use the Bcast that creates a tree-like communication pattern, that at each level the rank that receives the data can forward it to other ranks, reducing the overall communication time.
- $T(\text{CommHeader}) = 2 \cdot l \cdot \log p + \beta \cdot (m + n)$. Represents the communication cost for sending the chunk size and the real RecordHeader structures.
- $T(\text{CommIndex}) = l \cdot p + \beta \cdot n$. Represents the cost of communication for sending the original indexes.



Graph 2: Strong scaling execution time for `SIZE=10M`

As we can see from the Graph 2 of strong scalability, the implementation scales well up to 4 total cores, beyond which the performance degrades due to the overhead of synchronization and memory limits, in fact the ideal scaling is not achieved and also the time of execution increases.

On the other hand in Graph 3, we can observe that with different payload size the implementation works well, but with different size array the performance varies. The results indicate that as the payload increases, the performance remains relatively stable, this is very important because the performance must not be dependent from payload, demonstrating the effectiveness of the parallelization strategy employed. But as the array size increases, the time of executions tends to increase slowly, this trends is perfectly normal because the amount of data to process is larger, but the increase is not linear, indicating that the implementation works well and it have a good scalability.



Graph 3: Weak scaling - Thread number increase

5 Conclusion

In this work, we explored the performance implications of different parallelization strategies for file writing in a distributed environment. Our results indicate that while both OpenMP and FastFlow offer significant advantages over traditional sequential approaches, they each come with their own set of trade-offs.

The OpenMP implementation demonstrated superior performance in terms of I/O operations by leveraging true parallelism and efficient resource utilization. However, it also introduced complexity in terms of offset calculations and coordination among processes. In the sort part, the OpenMP implementation showed a bad performance with an execution time very high compared to the sequential version.

On the other hand, the FastFlow implementation provided a framework for parallel algorithms resulting in better performance in all cases. The combination of MPI and FastFlow allowed for improved execution times and scalability, reaching an execution time very fast considering also the time spent in communication and the final merge.

The implementation of MPI I/O was a crucial step in enhancing the performance of file writing operations. By enabling concurrent access to the file system, we were able to significantly reduce the time spent on I/O operations, which is often a bottleneck in distributed applications.

The future work will focus on further optimizing the write and load part to reduce execution time and use fully parallelization to minimize the time spent in IO part. Additionally, exploring hybrid approaches that combine the strengths of both OpenMP and FastFlow could yield even better performance for specific workloads.

The main goal of improving merge sort performance was achieved: the integration of MPI and FastFlow provided significant speedups in all tested configurations, mainly due to better load balancing among processes and reduced communication overhead in the merge phase.