



University of Pisa

Department Of Computer Science

Parallel and Distributed Systems: Paradigms and Models

Project:
Distributed out-of-core MergeSort

Student:

Abderrahmane Salmi

704608

a.salmi1@studenti.unipi.it

Professor:

Massimo Torquati

ACADEMIC YEAR 2024/2025

Chapter 1

Single Node Solution (Shared-Memory Parallelism)

1.1 Introduction

This project is focused on the design and implementation of a scalable, distributed, and out-of-core MergeSort for very large datasets that exceed the capacity of main memory. The sorting is performed on files that contain variable-sized records, each one contains a key (used for comparison) and a binary payload. The primary objective is to build a scalable MergeSort that works both on a single node (using shared-memory parallelism) and across multiple nodes (using distributed-memory models).

1.2 Shared-Memory Sorting with OpenMP

1.2.1 Overview

Given the memory limitation (e.g., $\leq 32\text{GB}$ RAM), the original input file is split into smaller chunks that can fit into memory, then these chunks are sorted independently in parallel using OpenMP, and finally merged into a single sorted output using a K-way merge.

1.2.2 Implementation Details

1. **Chunking Phase:**

The input file is divided into memory-sized chunks. Each chunk contains multiple records and is small enough to fit in memory.

2. **Parallel Sorting Phase:**

Each chunk is read into memory, parsed into Record objects, and sorted using `std::sort` with a custom comparator based on the record key. This phase is parallelized using OpenMP (`#pragma omp parallel for`), with each thread processing one or more chunks.

3. **Merging Phase:**

After sorting, each chunk is written to a temporary binary file. Once all chunks are sorted, a k-way merge is performed to combine them into a single sorted output file. This phase is

implemented using a min-heap (priority queue), where one record is read from each chunk and the smallest is selected and written to the output file. The process repeats until all chunks are handled and then the temporary chunk files are deleted (cleaned up).

To make sure the sorting is correct, a separate C++ utility code is used to verify that the output file is sorted correctly.

1.2.3 Challenges Encountered

Large File Handling

- We needed to handle large files that cannot fit entirely in memory.
- **Solution:** Split input file into small manageable chunks that fit into memory (bounded by a user-defined memory budget variable), sort them in parallel, and write them as sorted temporary files to disk.

Correct Parallelism

- Ensuring no race conditions or file I/O conflicts.
- **Solution:** Chunk processing is parallelized using OpenMP with file I/O managed independently per thread. Meaning each thread handles independent data and outputs to a separate temporary file.

Efficient Merging

- Merging sorted files without loading all the data into memory.
- **Solution:** Use k-way merge algorithm with priority queue.

1.3 Shared-Memory Sorting with FastFlow

1.3.1 Overview

The FastFlow implementation of merge sort was designed to efficiently handle large files that exceed the available memory by using task parallelism via the FastFlow farm pattern. We made it customizable by allowing the user to set the memory budget and number of worker threads.

1.3.2 Parallel Architecture: Farm Pattern

Our FastFlow implementation is developed using the farm pattern, which consists of:

- **Emitter:** Responsible for reading chunks from the input file and dispatching them to workers.
- **Workers:** Each receives a chunk, loads it into memory, performs an in-memory sort using `std::sort`, and writes the result to a new sorted temporary file.
- **Collector:** It collects the output file paths of the sorted chunks from the workers and keeps track of how many chunks have been processed.

This pattern allows us to fully parallelize the chunk sorting process across available CPU cores while keeping synchronization overhead minimal.

1.3.3 Implementation Details

1. Chunking Phase:

Input file is split into smaller chunks based on the user-specified memory budget. Those chunks are passed to the FastFlow emitter, which will be responsible for distributing them among workers.

2. Parallel Sorting Phase:

Each worker receives a chunk, parses it into records, sorts it in-memory, and writes the results to temporary files that are used later in the merging phase.

3. Merging Phase:

The merge phase is done sequentially (outside FastFlow) using a k-way merge since the number of sorted chunks is relatively small and the bottleneck is mostly I/O bound. A min-heap is used to merge all sorted files into a single output file.

1.4 Testing and Validation

1.4.1 Overview

To make sure that both the OpenMP and FastFlow implementations of external merge sort produces correct and identical results, we designed a testing pipeline that automates the following:

1. Input File Generation
2. Execution of Sorting Implementations (OMP and FF)
3. Output Verification
4. Comparing Outputs

1.4.2 Implementation Details

Step 1: Generating Test Files

We created a filegen utility code (tool) that is responsible for generating test files containing randomly generated records. The tool supports two methods:

- **gen_count**: generate a file with a fixed number of records.
- **gen_size**: generate a file approximately of a given size in MB.

Example:

Generate a test file with 1000 records

```
./filegen gen_count test_1000.bin 1000
```

Step 2: Sorting with OpenMP and FastFlow

The same input file is passed to both implementations:

```
./omp_mergesort sort test_1000.bin  
./ff_mergesort sort test_1000.bin
```

Each implementation produces its own output file:

```
test_1000_omp_output.bin  
test_1000_ff_output.bin
```

Both sorters are configured with the same memory budget and same number of threads to keep execution conditions equivalent.

Step 3: Output Verification

To ensure that the outputs are actually sorted, we use the `verify` command from `filegen`:

```
./filegen verify test_1000_omp_output.bin  
./filegen verify test_1000_ff_output.bin
```

This scans the file and checks that every record is less than or equal to the next. A "Verification PASSED" message confirms its correct.

Step 4: Comparing Outputs

Finally, we check whether both implementations produce identical sorted output. This is important to ensure that the sort is not only correct but also deterministically consistent between the two implementations.

```
./filegen compare test_1000_omp_output.bin test_1000_ff_output.bin
```

The tool compares every record in both files and reports the total number of matches, failing if there's even a single mismatch.

Automation via Makefile

All of the above was automated in the **Makefile** under the `make test` target. These run the full testing workflow, which makes it easy to validate the solution and make sure it works as expected.

Example (Test Run)

```

default@DESKTOP-SAP4292:~/projects/spm_project$ make test_large

=== Generating input files ===
./filegen gen_size large.bin 512
Generated file: large.bin (~512 MB)

=== OpenMP Sort ===
./omp_mergesort sort large.bin

OpenMP MergeSort initialized with 4 threads, memory budget: 256 MB
Starting OpenMP external merge sort...
Input: large.bin
Output: large_omp_output.bin

Phase 1: Creating sorted runs (parallel)...
Created 3 chunk files.
Thread 2: Processed 958699 records, temp file: ./temp_omp/run_2.tmp
Thread 0: Processed 1917396 records, temp file: ./temp_omp/run_0.tmp
Thread 1: Processed 1917396 records, temp file: ./temp_omp/run_1.tmp
Phase 1 completed in 14.4604 seconds
Merged 4793491 records into output file

OpenMP External Merge Sort statistics:
Total records processed: 4793491
Phase 1 time: 14.4604 seconds
Phase 2 time: 0 seconds
Total elapsed time: 18.3344 seconds
Sort completed in 23.7414 seconds

Verification PASSED! Total records: 4793491

=== FastFlow Sort ===
./ff_mergesort sort large.bin

Input File      : large.bin
Output File     : large_ff_output.bin
Memory Budget   : 256 MB
Worker Threads  : 4

FastFlow External MergeSort starting...
Memory budget: 256 MB
Workers: 4
Created 3 chunks
FastFlow Worker: Sorted chunk 2 (958699 records) -> temp_ff_87616/sorted_chunk_2.bin
FastFlow Worker: Sorted chunk 0 (1917396 records) -> temp_ff_87616/sorted_chunk_0.bin
FastFlow Worker: Sorted chunk 1 (1917396 records) -> temp_ff_87616/sorted_chunk_1.bin
Sorted 3 chunks in parallel
Merged 4793491 records into output file
Merged chunks into final output: large_ff_output.bin

Sort completed in 20.2459 seconds.
Verification PASSED! Total records: 4793491
Verification: PASS

=== Compare Results ===
./filegen compare large_omp_output.bin large_ff_output.bin
File comparison PASSED! Records compared: 4793491

```

Chapter 2

Multiple Nodes Solution (using distributed-memory models)

2.1 Overview

This section describes the implementation of the distributed sorting algorithm using MPI (Message Passing Interface) for inter-process parallelism and OpenMP for intra-process parallelism. The goal is always to efficiently sort large files that exceed the memory capacity of a single machine, using multiple nodes or cores.

2.2 Implementation Details

The MPI sorting solution has four main stages:

2.2.1 File Partitioning

The input file is partitioned based on byte offsets, dividing it into approximately equal-sized chunks, and each MPI process (rank) is assigned a chunk to process. To prevent records from being split across ranks, we try to align partition boundaries with complete record boundaries, in other words, we don't want to have a chunk that contains an incomplete record, so we keep processing the file until the record is complete, then we can safely partition a chunk. The "calculate_file_partition" function is responsible for this.

This ensures:

- All records are processed exactly once.
- No record is split or corrupted during I/O.
- Each rank operates independently on its chunk.

2.2.2 Local Sorting with OpenMP

After partitioning, each rank:

- Reads its assigned records

- Writes them to a local temporary file
- Sorts the file using our existing OpenMP solution.

The sorted results are then reloaded into memory for the distributed merge.

This step is extremely parallel, because each rank works on its own subset independently, and each rank uses OpenMP to efficiently sort using multi-threaded solution.

2.2.3 Distributed Merge via Sample Sort

To produce a globally sorted output, a distributed merge using a sample sort algorithm is used. The key steps are:

- **Sampling:** Each rank selects a small number of representative keys from its sorted data.
- **Gathering Samples:** Samples are collected at rank 0 using `MPI_Gatherv`, which aggregates variable-sized data from all ranks.
- **Splitter Selection:** Rank 0 sorts all gathered samples and selects size - 1 splitter keys to divide the key space evenly. These splitters are then broadcasted to all ranks.
- **Partitioning:** Each rank partitions its data, where each partition corresponds to the data destined for a specific rank.
- **All-to-All Exchange:** Ranks exchange partitions using `MPI_Alltoallv`, ensuring each rank receives all records that belong to its final key range.
- **Final Local Merge:** Each rank merges the received records and writes its portion of the sorted data to a rank-specific output file.

2.2.4 Output Generation

Each rank writes its sorted data to a separate output file, then rank 0 concatenates the files in rank order to form the final output file. This approach ensures scalability, because it avoids a central bottleneck in writing the final output.

Temporary files created during the sort are cleaned up after use.

Chapter 3

Performance Evaluation

3.1 Overview

This section presents a detailed analysis of the performance of our parallel sorting implementations using FastFlow and OpenMP. The goal is to evaluate how the solutions scale when we change:

- Number of threads/workers (1, 2, 4, 8, 16)
- Input size (number of records: 10M, 50M)
- Payload size (64B, 512B)

The experiments were executed on a single node of the SPM cluster.

3.2 Datasets Used

Filename	Records (N)	Payload (Bytes)	Size	Category
data_10M_p64.bin	10M	64	~725 MB	Small N, Small Payload
data_10M_p512.bin	10M	512	~4.9 GB	Small N, Large Payload
data_50M_p64.bin	50M	64	~3.6 GB	Large N, Small Payload
data_50M_p512.bin	50M	512	~25 GB	Large N, Large Payload

3.3 Methodology

Each dataset was sorted using both the FastFlow and OpenMP implementations. We measured the execution time for each run and computed the speedup and efficiency based on the single-thread result.

$$S(p) = \frac{T_{\text{seq}}}{T_{\text{par}}(p)} \quad (3.1)$$

$$E(p) = \frac{S(p)}{p} = \frac{T_{\text{seq}}}{T_{\text{par}}(p) \times p} \quad (3.2)$$

Where:

- $S(p)$: Speedup
- $E(p)$: Efficiency
- T_{seq} : Execution time with 1 thread
- $T_{par}(p)$: Execution time with p threads/workers

3.4 Execution Time

FastFlow Execution Time (s)					
Dataset	1W	2W	4W	8W	16W
data_10M_p64	27.39	24.52	17.68	16.82	17.42
data_10M_p512	55.72	37.09	31.74	28.70	30.42
data_50M_p64	133.96	105.32	89.25	88.63	95.40
data_50M_p512	347.01	306.78	248.09	220.37	246.67

OpenMP Execution Time (s)					
Dataset	1T	2T	4T	8T	16T
data_10M_p64	29.04	24.56	20.81	19.43	18.58
data_10M_p512	47.89	39.12	33.27	29.23	31.01
data_50M_p64	148.21	124.93	103.47	97.33	99.89
data_50M_p512	544.01	409.98	342.87	311.94	319.47

Table 3.1: Execution times (in seconds) for FastFlow and OpenMP across varying workloads (W = workers, T = threads).

3.5 Speedup and Efficiency Analysis

The following plots summarize the performance across all datasets for both fastflow and openmp:

3.6 Observations

For the smaller datasets (data_10M_p64.bin and data_10M_p512.bin), both FastFlow and OpenMP show significant speedup when increasing the number of workers/threads from 1 to 8. This indicates that the parallelization strategy is effective for these sizes, and there is a good CPU-bound parallel scalability. However, as the number of workers/threads increases more, the speedup decreases, which means that the overhead of managing more threads/workers starts

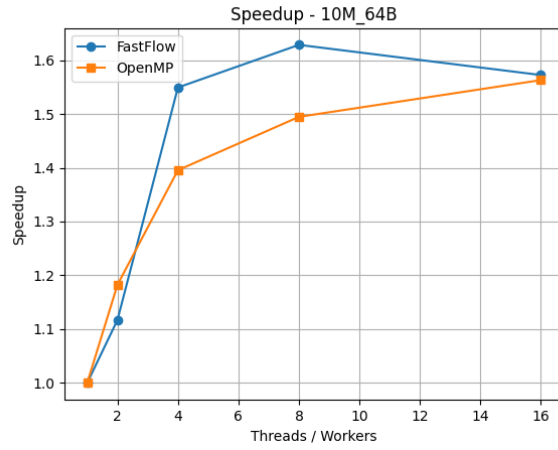


Figure 3.1: Speedup - 10M Records, 64B Payload

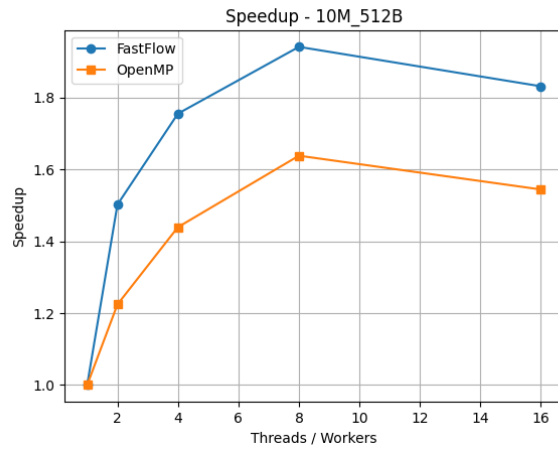


Figure 3.2: Speedup - 10M Records, 512B Payload

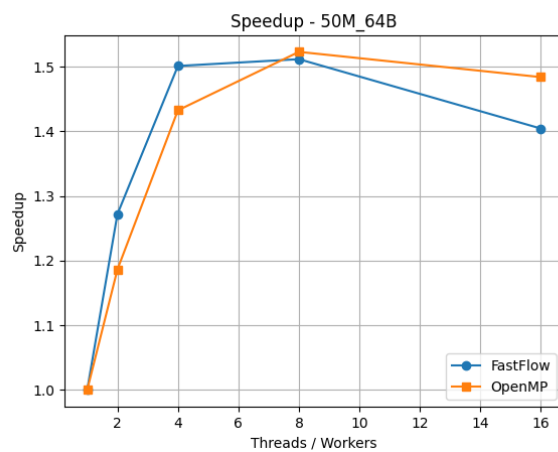


Figure 3.3: Speedup - 50M Records, 64B Payload

to outweigh the benefits of parallel execution. A possible reason for this decrease can be the increasing influence of I/O operations and memory bandwidth limitations.

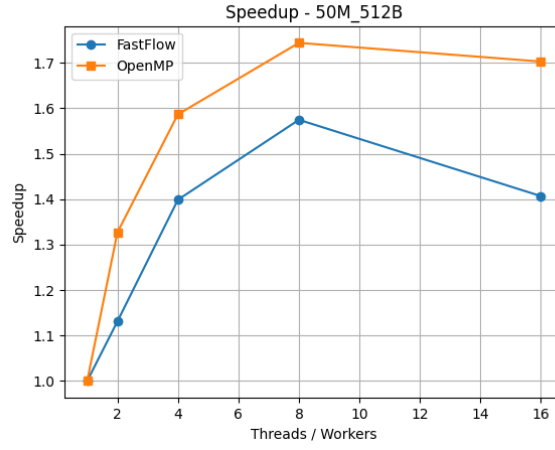


Figure 3.4: Speedup - 50M Records, 512B Payload

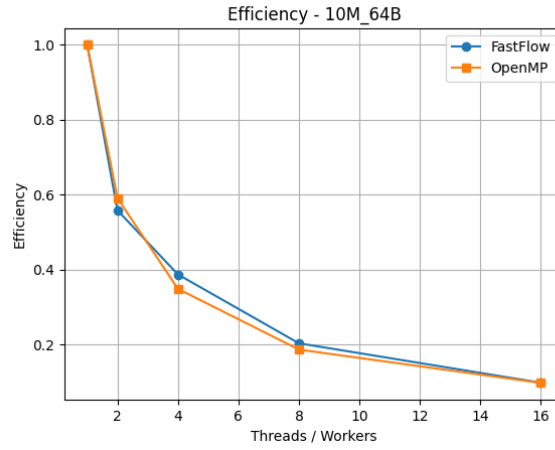


Figure 3.5: Efficiency - 10M Records, 64B Payload

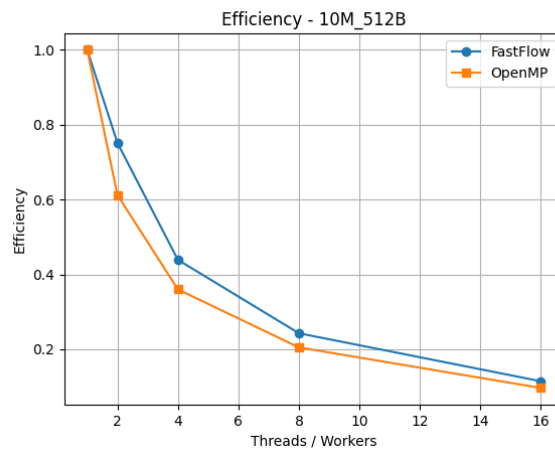


Figure 3.6: Efficiency - 10M Records, 512B Payload

For the larger datasets (data_50M_p64.bin and data_50M_p512.bin), the speedup is less pronounced, especially for higher numbers of workers/threads. This can be attributed to increased memory contention and I/O overhead, which become more significant as the problem

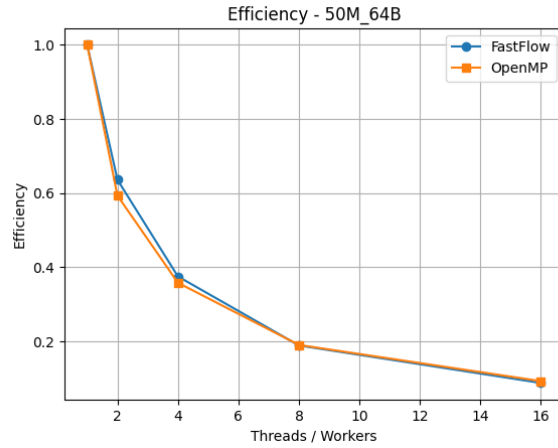


Figure 3.7: Efficiency - 50M Records, 64B Payload

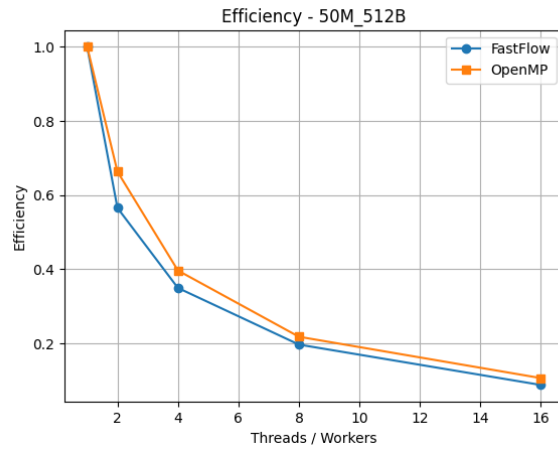


Figure 3.8: Efficiency - 50M Records, 512B Payload

size grows. The larger datasets require more memory and I/O operations, leading to increased contention for these resources among the workers/threads. This contention can reduce some of the benefits of parallelism, resulting in less significant speedup improvements.

Another important observation is that the payload size significantly affects performance. For 64-byte records, sorting is more compute-bound, and both systems scale well up. On the other hand for 512-byte records, the system becomes more memory and I/O bound, because read and write operations to disk dominate, especially during the merge phase, which reduces the benefit of added parallelism.

Chapter 4

Cost Model and Analysis

4.1 Approximate Cost Model

The total runtime T_{total} of the distributed solution on a single node with shared-memory parallelism can be approximated as the sum of three main phases:

$$T_{\text{total}} \approx T_{\text{chunk}} + T_{\text{sort}} + T_{\text{merge}}$$

- **Chunking time** (T_{chunk}): It is the time to read the input file and split it into smaller chunks. This phase is I/O bound, and its cost is proportional to the input file size N :

$$T_{\text{chunk}} = \alpha \cdot \frac{N}{B_{\text{disk}}}$$

where we assume α is an overhead factor (which can depend on buffering, system, etc.) and B_{disk} is disk bandwidth.

- **Sorting time** (T_{sort}): It is the time to sort all the chunks in parallel. If we assume we have P parallel threads and chunks of size M , the sorting of each chunk will cost approximately $O(M \log M)$. So the total cost will be:

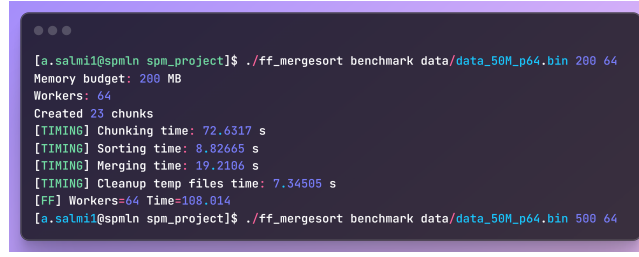
$$T_{\text{sort}} \approx \frac{N}{P} \log M$$

because the workload is evenly distributed.

- **Merging time** (T_{merge}): It is the time to merge all the sorted chunks. It is also dominated by I/O operations. The cost is roughly:

$$T_{\text{merge}} = \beta \cdot \frac{N}{B_{\text{disk}}} + N \log k$$

where k is the number of chunks, and β represents the sequential disk reads/writes. And since the merging is mostly I/O bound, $\beta \cdot \frac{N}{B_{\text{disk}}}$ will dominate.



```

[a.salmi@spmln spm_project]$ ./ff_mergesort benchmark data/data_50M_p64.bin 200 64
Memory budget: 200 MB
Workers: 64
Created 23 chunks
[TIMING] Chunking time: 72.6317 s
[TIMING] Sorting time: 8.82665 s
[TIMING] Merging time: 19.2106 s
[TIMING] Cleanup temp files time: 7.34505 s
[FF] Workers:64 Time:108.014
[a.salmi@spmln spm_project]$ ./ff_mergesort benchmark data/data_50M_p64.bin 500 64

```

4.2 Bottleneck Phases

- **Disk I/O:** Both chunking and merging phases are I/O-bound, limited by the read/write bandwidth of the disk subsystem. And in our solution, I noticed that the chunking is the phase that takes the longest each time, so we can say its one of the biggest bottlenecks we have in this system.
- **Multi-Node Bottleneck:** MPI all-to-all communication feels like it is sensitive to network bandwidth and imbalance in partition sizes (due to variable payloads), so when the payload increases, the traffic volume increases, so the MPI time grows.

4.2.1 Computation-to-Communication Overlap Effectiveness

Our MPI solution is used to distribute the sorting workload across multiple processes, where each rank reads and sorts its own partition of the input file using OpenMP for parallelism locally.

The key to efficient scaling is overlapping computation (which is sorting) with communication (which is exchanging sorted data during merge).

Lets define:

$$T_{\text{comp}} : \text{Local sort} + \text{final merge per rank} \quad , \quad T_{\text{comm}} : \text{MPI all-to-all data exchange}$$

If we take results from applying the solution to the test file that has $50\text{M} \times 512\text{B}$, we find:

$$T_{\text{comp}} \approx 60 \text{ s}, \quad T_{\text{comm}} \approx 20 \text{ s}, \quad \frac{T_{\text{comm}}}{T_{\text{total}}} \approx 25\%$$


Which means our MPI overhead is non-negligible but acceptable.

I noticed that for smaller payloads communication is faster, so more time is spent sorting than transmitting.

4.2.2 Challenges Encountered

One of the biggest challenges I encountered was reducing the time of chunking the input file, because I noticed that it was the dominante time each time, for example I have the following run:

After applying several optimizations, the chunking time was dramatically reduced, as shown here:



```
[a.salmi@spm1n spm_project]$ ./ff_mergesort benchmark data/data_50M_p64.bin 200 64
Memory budget: 200 MB
Workers: 64
Created 23 chunks
[TIMING] Chunking time: 7.85713 s
[TIMING] Sorting time: 3.21373 s
[TIMING] Merging time: 0.719634 s
[TIMING] Cleanup temp files time: 3.20363 s
[FF] Workers=64 Time=14.9944
```

4.2.3 Optimizations Adopted

Here are few of the key optimizations that were implemented to improve the chunking phase:

- **Block-wise Chunking with Alignment:** Rather than reading the input file record-by-record, the file was divided into fixed-size blocks. Each block boundary was aligned to record boundaries by scanning forward up to 1 KB, minimizing partial records. This approach significantly reduced disk I/O overhead and parsing complexity.
- **Parallel Chunk Extraction:** The chunk creation process was parallelized using `std::async` with synchronized access to a shared input stream. This allowed effective utilization of multi-core CPUs and SSD parallelism to increase throughput.
- **Single File Open:** The input file is opened once and shared across all threads, avoiding repeated open/close operations and reducing file system contention.
- **Robust Alignment Strategy:** The alignment method was designed to gracefully handle edge cases such as very small or malformed files.