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 (OpenMP and FastFlow)

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., <= 32GB 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 first memory-mapped using mmap() for fast access. Then, the file is scanned sequentially to generate logical chunks (offsets), which are carefully computed so that record boundaries are not broken (all chunks contain complete records). The estimated chunk sizes are computed based on the total input size and available memory budget. Finally, the logical chunks are given to OpenMP threads where each thread creates a file for each chunk, which is done in parallel.

2. **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), where each thread processes one or more chunks.

After sorting, each sorted chunk is written to a temporary file to be used later in the merge.

3. **Merging Phase**:
   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 handeled and then the temporary chunk files are deleted (cleaned up).

PS: The entire sorting function is timed using "omp_get_wtime()" for accurate performance measurement. PS: 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

Trying to make the chunking and merging phases parallel and efficient was one of the most difficult parts of the implementation. Different strategies were considered, and some were tried (tested) and discarded due to performance issues or implementation complexity.

Chunking: For example, for the chunking, two parallelization strategies were considered:

- **Strategy A: Chunking before sorting**
  First, a sequential pass scans the file and computes all logical chunks. Then, multiple threads write each chunk to disk in parallel, then when all chunks are written, they are passed to the sorting function. This approach was effective and simple, and it was the one used in the project.
- **Strategy B: Chunking and sorting at the same time**
  Same start with a sequential scan of the file and computeing all logical chunks, then each thread performs chunking and immediately launchs an OpenMP task to sort the chunk, this way we don't wait for all chunks, each ready chunk is sorted immediatly. Although this idea is great in theory, in practice it introduced more overhead and was slower than the first one.

Merging:

The merging phase also posed a design challenge. The goal was to reduce total merge time by parallelizing it. I tried implementing a multi-stage parallel merge, where groups of chunk files were merged in parallel stages, and the intermediate files were merged again until only one final file remained.

While this would work well when the number of chunks is very large, it introduced too much I/O overhead in our project and test files. At the end, the multi-stage merging became slower than the simple sequential k-way merge.

1.3 Shared-Memory Sorting with FastFlow

1.3.1 Overview

Just like OpenMP, This solution implements a parallel external merge sort on large binary files that cannot fit entirely in memory, but it uses the FastFlow framework to do so.

1.3.2 Parallel Architecture: Farm Pattern

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

- **Emitter**: Responsible for reading and splitting the input file into smaller chunks. Just like we did in OpenMP, we scan the file to determin the logical chunks and make sure we don't divide any records in the middle, then we pass those chunks to the workers.
- **Workers**: A pool of parallel workers that sort each chunk independently. Each worker receives a chunk, parses it into records, sorts the records in-memory using `std::sort`, and writes the sorted data to its temporary output file.
- **Collector**: Recieves the sorted chunks from the workers and performs a final k-way merge to produce a single output file. Temporary files are then deleted after the merge is complete.

PS: The total time for the sorting and merging phases is measured using the fastflow utility time functions for better performance accuracy.

Chapter 2

Multi-Node Hybrid Solution (MPI + FastFlow)

2.1 Overview

This section describes the implementation of the distributed sorting algorithm using MPI (Message Passing Interface) for inter-process parallelism and FastFlow 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.

The solution has two modes:

- **Single-node mode** (when MPI size = 1): the file is sorted locally on the only node available using the FastFlow sorter.
- **Distributed mode** (MPI size > 1): rank 0 becomes the coordinator, and all other ranks act as sorting workers.

2.2 Implementation Details

My solution was inspired from the farm pattern like we did in FastFlow, where we have 2 special nodes that are the emmiter and the collector, and the rest are workers. Same logic applies here, rank 0 is the coordinator, which is the equivalent of both the emitter and collector (does both their jobs), and the rest of the ranks are workers.

Rank 0 - Coordinator:

- Logically splits the input file into chunks, making sure the record boundaries are preserved. Same chunking logic that was used in FastFlow and OpenMP is also used here.
- Writes each logical chunk to a physical file on disk.
- Distributes chunk files to workers in a round-robin way.
- Collects the paths to the sorted chunk files from each worker.
- Performs a final k-way merge to produce the fully sorted output.

- Deletes all temporary files after merging.

Other Ranks - Workers:

- Receives the list of chunk files assigned by the coordinator.
- For each file, uses the existing FastFlow code to sort it.
- Sends back to the coordinator the paths to the sorted output files.

PS: The MPI sorting solution is timed using `MPI_Wtime()` for more accurate performance measuring.

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, 32, 64)
- Input size (number of records: 10M, 50M)
- Payload size (64B, 512B)

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.

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

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

Where:

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

3.4 Speedup and Efficiency

Sequential Execution Time (s)	
Dataset	**Time (s)**
`data_10M_p64`	52.2425

Table 3.1: Sequential execution time (in seconds)

Table 3.2: OpenMP Performance

Threads	Time	Speedup	Efficiency
2	37.5752	1.390345	0.70
4	32.4526	1.609809	0.40
8	26.1782	1.995649	0.25
16	24.0571	2.171604	0.14
32	20.2746	2.576746	0.08
64	20.4024	2.560606	0.04

Table 3.3: FastFlow Performance

Workers	Time	Speedup	Efficiency
2	30.6810	1.702764	0.85
4	18.4251	2.835398	0.71
8	17.5604	2.975018	0.37
16	16.7573	3.117597	0.19
32	20.5539	2.541732	0.08
64	20.3824	2.563118	0.04

Table 3.4: MPI + FastFlow Performance

MPI Procs	Time	Speedup	Efficiency
2	26.9136	1.941119	0.97
4	13.9847	3.735690	0.93
8	11.9277	4.379931	0.55

And here are the plots for the previous tables:

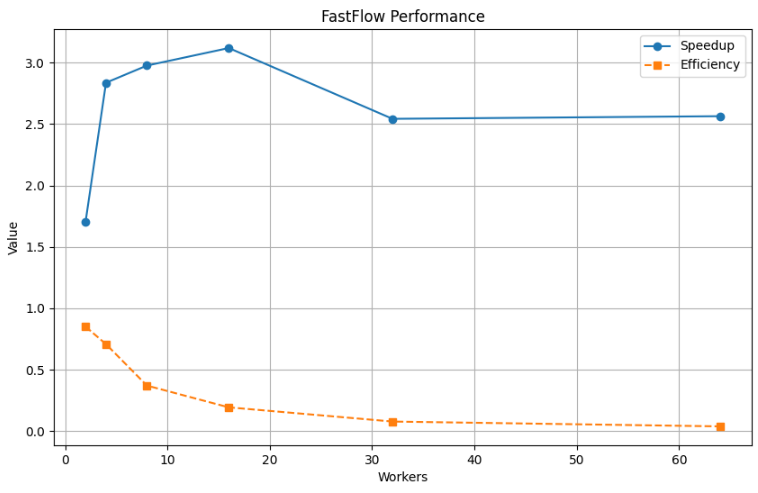

Figure 3.1: FastFlow Performance

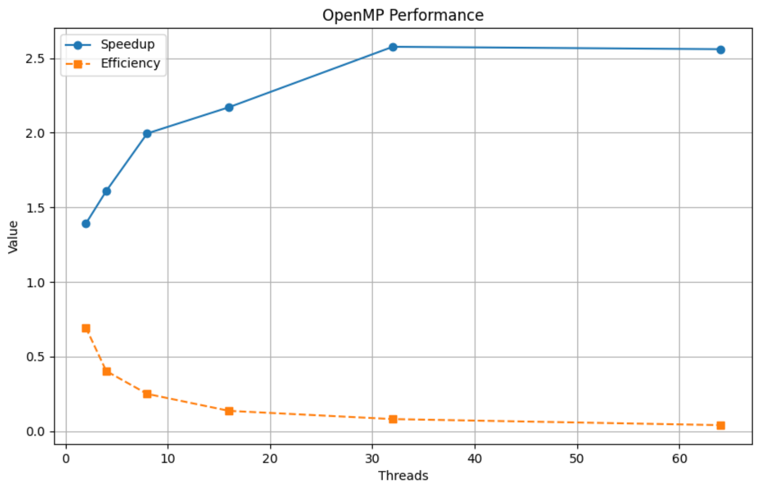

Figure 3.2: OpenMP Performance

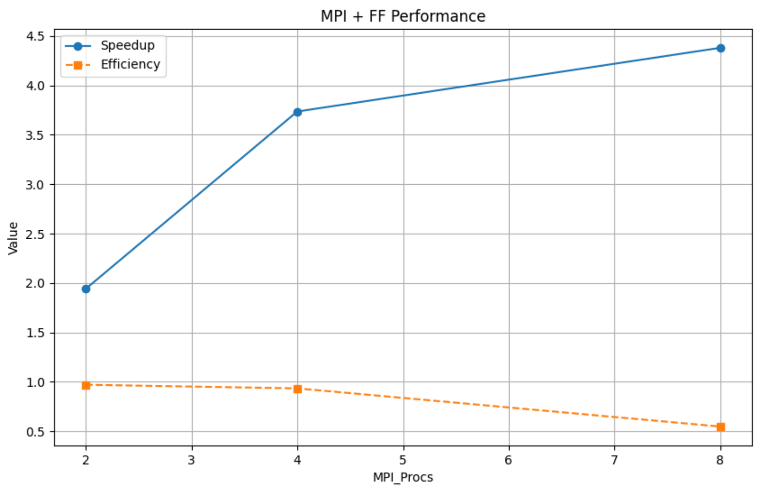

Figure 3.3: MPI Performance

3.5 Observations

Both FastFlow and OpenMP show significant speedup when increasing the number of workers/threads. This indicates that the parallelization strategy is effective for these sizes, and there is a good CPU-bound parallel scalability. However, sometimes as the number of workers/threads increases more, the speedup decreases, which means that the overhead of managing more threads/workers starts 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.

Chapter 4

Cost Model and Analysis

4.1 Cost Model Approximation

The distributed external merge sort can be modeled using a combination of the **BSP** and **Farm** cost models that we saw in the course, because they represent well the hybrid architecture of MPI + FastFlow.

Let:

- p = number of MPI processes (1 coordinator + $p - 1$ workers)
- t = number of FastFlow threads per worker
- n = total number of records
- k = number of chunks
- M = total memory budget
- g = communication cost per word
- l = MPI synchronization latency

We can estimate the total cost as:

$$T_{\text{total}} = T_{\text{chunking}} + T_{\text{distribution}} + T_{\text{local-sort}} + T_{\text{merge}} + T_{\text{cleanup}}$$

1. **Chunking (Coordinator only)**:

$$T_{\text{chunking}} = O(n) \quad \text{(sequential scan + disk I/O)}$$

2. **Distribution (MPI communication)**:

$$T_{\text{scatter}} = k \cdot (t_0 + d \cdot s)$$

   where t_0 is startup latency, d is size of data, s is per-byte transmission cost ($\approx 1/B$ where B is the bandwidth)

   This is inspired from the simple communication cost model seen in the course.

   In our implementation, the coordinator sends just file paths, so T_{scatter} should be very small.

3. **Local Sort (Workers)**:

   Since each worker is sorting using FastFlow, and our FastFlow is using the farm pattern, so T_{sort} is just $T_{\mathrm{c}}^{\mathrm{farm}}(worker\_chunks, k)$

   $$T_{\mathrm{sort}} = T_{\mathrm{c}}^{\mathrm{farm}}(worker\_chunks, k) = (t+1) \cdot T_{comm} + (worker\_chunks/t) \cdot (T_{\mathrm{s}}^{\mathrm{w}} + T_{comm})$$

   where t is the number of FastFlow workers (threads), $worker\_chunks$ is the number of chunks a rank worker has to process, and $T_{\mathrm{s}}^{\mathrm{w}}$ is worker service time, which is $\max(T_{\mathrm{seq}}, T_{\mathrm{comm}})$

4. **Merge (Coordinator)**:

   Let k be the number of sorted chunks. The final k-way merge at the coordinator is:

   $$T_{\mathrm{merge}} = O(n \cdot \log k) + O(\mathrm{I/O})$$

5. **Cleanup:**

   Removing temporary files is done sequentially and its cost is negligible in most scenarios.

4.2 Bottleneck Phases

The most notable bottlenecks are:

- **Chunking**: Since its only performed by the coordinator (rank 0), it can be a performance limiter on very large files.
- **Final Merge**: Same as chunking, since it's only handled by rank 0, it can be a bottleneck. Although we considered a multi-stage parallel merge, but it introduced too much I/O and coordination overhead for this specific project.
- **File I/O**: I/O in general always remains a bottleneck especially when the memory budget is low, since it'll result in a larger number of chunks, which will result in too many I/O operations that will slow down the whole solution.

4.2.1 Computation-to-Communication Ratio

We define the computation-to-communication ratio (γ) as:

$$\gamma = \frac{\alpha}{\beta}$$

Where:

- α = total CPU work (chunking, sorting, merging)
- β = communication overhead (MPI messages, file path exchange, coordination)

In our case, γ is high, because:

- Communication usually involves only file paths, which are lightweight, no big data is exchanged.
- Each worker operates independently and locally, no communuctaion between workers.
- Communication only occurs in setup and final gather phase.

Conclusion: The high γ value indicates good scalability, meaning the system is performing significantly more useful work than communication overhead.