Universita di Pisa

Parallel and Distributed Systems

Professor Massimo Torquati

Project: Distributed Out-of-Core Merge Sort

github:https://github.com/bhuiyan-chafi/spm/tree/main/spm-sort

By: ASM Chafiullah Bhuiyan

Computer Science and Networking

Academic Year: 2025-2026

1 Abstract

In this project we tried to solve the problem of sorting large datasets that cannot fit in memory. The problem appears often in databases, search engines, and data processing systems. We must adopt external sorting techniques for datasets that don't fit in MEMORY. These techniques read data in segments, sort each segment in parallel, and then merge them on disk. In my implementation I have started with a function which checks the input size from the disc and takes decision if it can be operated `IN_MEMORY` or `MEMORY OUT-of-CORE`. In the `OOC` mode I have segmented the data into smaller parts and each segment is bounded by a constant called `DISTRIBUTION_CAP`. The program sorts each segment and writes back on disk. Finally, it performs a k-way merge to produce the sorted output(one big binary file). For the implementation I have used structured parallel programming concepts from the course`[FastFlow, OpenMP, MPI]`. Both solutions follow the same pattern where an emitter stage reads the input and creates segments. Each segment is divided into DEGREE** slices. Each slice fits in the L1 cache of the `PEs`. Worker stages fetch these slices and sort them in parallel following the map-style data parallelism. If the dataset doesn't fit in memory, workers(in FastFlow) and writers(in OpenMP) writes those intermediate segments on disc and then the Collector finally merges them back into one `output.bin` file.

1.1 The Workflow

The implementation of OpenMP uses a manual thread management system with SafeQueue and mutex locks for synchronization. To write the intermediate results a thread-pool has been used with a tunable queue size. The formula $DEGREE * WORKERS$ has been used to generate number of tasks for `dynamic load distribution`, so that the workers can fetch tasks whenever they are free. This also puts back-pressure to the emitter when all the workers are busy processing tasks. On the other hand **FastFlow** provides **set_scheduling_ondemand()** for automatic load balancing. In this mode workers pull tasks when they are ready. The back-pressure is handled by FastFlow's **internal bounded channels** between emitter and workers. When all workers are busy their input queues become full and emitter's **ff_send_out** blocks them automatically. This slows down the reading from the file. **The FastFlow runtime manages these channels internally which removes the need for manual queues, condition variables, and queue size.** This makes the implementation much easier for the developer. The final `multi-node` version is built with **MPI+FastFlow** following a `master-worker` pattern. The master process (rank 0) distributes segments to worker nodes. Each worker node runs a FastFlow farm internally. After sorting, workers send results back to the master. The master performs the final global merge.

1.2 Encountered Problems

Even though I have discussed the problems in detail later, I will state the major ones here. The first naive problem was defining a **vector<uint_8> payload** for the items instead of **pointer**. Professor specifically mentioned **char payload[]** for the payload, but I defined vectors as it gives flexibility for dynamic sized arrays. The result was catastrophic, and I have changed it in the later versions of OpenMP implementation(there are 6 versions of OpenMP; with each upgrade one major issue has been solved). One among the other problems was poor synchronization between the **emitter** and **workers**. The synchronization was so bad that the emitter kept reading from the input while the workers were busy with their existing tasks. Another problem was having a bad distribution system which affected the performance too much. The last major problem was putting too much pressure on the collector(in memory out-of-core phase) that it was almost paralyzed. There were other problems of poor communication between nodes in the multi-node version, but I have discussed that in a separate section later.

1.3 Future Work

Right now the FastFlow gives similar performance like OpenMP but I think with more optimization and tests the results can be improved. For example, we can suggest an efficient algorithm to choose `DISTRIBUTION_CAP` and `DEGREE` based on input size and system configuration. We can also try lock-free mechanism for the safe-queues. The current implementation uses array of structures which can be replaced with structures of array and tested.

[**] a tunable CONSTANT that can be tuned based on input size, data-type and hardware capacity. It generates tasks based on the parameters so that each worker always process inputs equal to it's private cache.

Right now emitter and collector stages are sequentially(discussed later in brief) reading and writing the data in/out from/to a file, which can be parallelized and tested.So, we can say that this implementation is not 100% efficient but it is a good starting point which covers most of the concept and coding practices from the course.

2 Prerequisites

As per the project requirements I have implemented two versions, **OpenMP and FastFlow for single node; MPI and FastFlow for multi-node.** Before discussing the implementation I will discuss some pre and post requisite programs. Data generation and verification are two most important part of the whole project. Since they are time consuming, to save time I have used pre-generated data by using the program **generator.cpp**(/src/generator.cpp) and to verify the data I have used **verify.cpp & reader.cpp**. Let us discuss them first very shortly. Also a short demonstration is provided in the `readme.md` file of the github repository.

2.1 Data Generation

The program follows the data-structure provided by my professor in the project. It generates arbitrary number of records defined by the user. For the dynamic number of payload I have used a class **CompactPayload**, which uses `uint_8` pointer for data storage. Then I have used two `struct` for storing and processing the records: **struct** Record{**uint64_t** key; **uint32_t** len;} which stores the 64-bit key followed by the 32-bit payload length. After reading from disk, the program converts records into **struct** Item{**uint64_t** key; **CompactPayload** payload;} for in-memory processing. It stores the length **uint32_t** at the beginning of the allocated memory block, followed by the actual payload bytes. The program uses `uniform_int_distribution()` for random data generation and stores them as binary file ***.bin**. The execution of all programs are described separately in the **Execution** section.

2.2 Data Verification

The program **verify.cpp** ensures data integrity and correctness of the sorting process using a multi-step validation approach. Instead of comparing each record key for correct sorted order, I have followed a divide & conquer technique followed by computing a global hash. First, it computes a cumulative hash of all payload data from the input file using the FNV-1a 64-bit hash algorithm. This creates a unique fingerprint of the input data. Second, it performs basic file validation by checking that the input and output files have identical sizes and contains the same number of records. If either check fails, the verification aborts immediately. Third, the program divides the output file into equal-sized chunks based on the number of available workers. Each chunk is loaded into memory and verified independently. For each chunk, the program checks local sorted order by ensuring every key is greater than or equal to the previous key within that chunk. It also verifies global sorted order by comparing the first key of each chunk with the last key of the previous chunk. While checking the sorted order, the program simultaneously computes payload hashes for all records and accumulates them. Finally, the cumulative output hash is compared with the input hash. If both hashes match and all sorted order checks pass, the verification succeeds, confirming that the output is correctly sorted and no data was corrupted or lost during the sorting process.

The reader program **reader.cpp** provides a simple visual inspection tool for verifying sorted order by printing a specified number of records from either the input or output file. The user specifies how many records to display *e.g.*, 100, 1000, and the program sequentially reads and prints each record's index, key value, and payload length. By examining the printed key values in order, users can visually confirm whether the keys are in non-decreasing order *sorted* or in random order *unsorted*. This manual inspection complements the automated verification program and helps during development and debugging to spot sorting issues quickly.

2.3 Execution Commands

2.3.1 Data Generation

The program takes arbitrary number of parameters. The first command is the program name followed by payload length(8 to 256 bytes long) and number of records(1M/5M/10M or 100M). To execute use the following:

```
$ cd "project-root"
$ cd src/
$ make generator
$ ./generator 256 1M
```

The program generates the records under **/data/rec_*_*.bin** file. In the programs this directory has been hard-coded for both input and output.

2.3.2 Data Verification

The program takes the name of the input file only. The output file **(/data/output.bin)** is universal for all type of inputs. Which means if we run two programs one after another with two different input, we have verify one before running the another one. One useful could be:

```
$ cd "project-root"
$ cd src/
$ make verify
$ ./verify rec_1M_256.bin
```

2.3.3 OpenMP

The program has 6 versions, each with different problems(discussed later). But the final version **openmp.cpp** has been finalized with all the fixes and some optimizations. To run the program use the following command:

```
$ cd "project-root"
$ cd src/
$ make openmp
$ ./openmp 1M 256 4 32
```

The first parameter **program_name** is followed by **number of records**, **payload length**, **number of workers** and **MEMORY_CAP**. **MEMORY_CAP** is an important parameter which follows the **memory out of core operation** mentioned in the project description. Actually it is very hard to test a file >32GiB in the **spmcluster**, that is why this parameter is tunable to test the behavior. If you have a file of 4GiB and you want to test if the program stops with a memory of 2GiB or not! You can simply tune the **MEMORY_CAP** and set it to any size less than 4GiB and test the behavior.

2.3.4 The Mapping String

Before running FastFlow programs, my professor instructed us to execute `mapping_string.sh` script to configure optimal thread-to-core mapping for the specific machine topology. The script uses **hwloc** tools to detect the system's CPU architecture including physical cores, logical cores, and NUMA nodes. It builds a mapping string that assigns threads to cores in topologically contiguous order rather than following the operating system's default numbering. For example, instead of sequential OS numbering like 0,1,2,3, it creates a string like 0,4,2,6,1,5,3,7 which groups cores by NUMA nodes and fills physical cores before hyperthreads. This mapping is written to FastFlow's `config.hpp` file along with the total number of logical and physical cores available. Running this script improves performance significantly because it ensures NUMA-aware thread placement which reduces cross-NUMA memory access latency, optimizes cache utilization by keeping threads on the same physical core together, prevents the OS scheduler from arbitrarily migrating threads across cores which causes cache thrashing, and intelligently handles hyperthreading by filling physical cores first before using hyperthread contexts. Without this configuration, the operating system might place communicating threads on different NUMA nodes causing 2-3x slower memory access, or move threads between cores destroying cache locality. This topology-aware placement improves FastFlow performance to some level on NUMA systems, which is critical for the tight communication patterns between emitter, workers, and collector in the farm pipeline where threads frequently exchange data through bounded channels.

2.3.5 Sequential

Number of workers is unused here. But still we have to put a number here only for parsing the `cli` arguments.

```
$ cd "project-root"
$ cd src/
$ make sequential
$ ./sequential 1M 256 1 32
```

2.3.6 FastFlow

The FastFlow version has a similar execution command like OpenMP with same number and type of parameters:

```
$ cd "project-root"
$ cd src/
$ make farm
$ ./farm 1M 256 4 32
```

2.3.7 MPI+FastFlow

This is the final version of the project. The program execution is also similar but I recommend to use **-np > 1**. It is not mandatory but I have used a Master-Worker pattern, where RANK 0 is the Master and RANK 1+ are Workers. To execute the program use the following command:

```
$ cd "project-root"
$ cd src/
$ make mpiff
$ mpirun -np 2 ./mpiff 1M 256 2 32
```

2.3.8 Simulation Scripts

It is also possible to run all the programs with arbitrary number of inputs and workers. Based on the number of input types I have designed 4 different scripts, which runs 4 different number of record inputs with 2 4 8 16 32 48 64 128 number of workers. The scripts can be run using the following commands:

```
$ cd "project-root"
$ cd src/
$ bash run_1M_256.bin 32
```

The last parameter is the **memory cap**, you can put your desired number to test the behavior. If you are using **spmcluster.unipi.it**, make sure you are using SRUN commands. To facilitate that you can use the following command:

```
$ cd "project-root"
$ cd src/
$ bash run_simulation.sh 1M_256 32
```

Here the parameters are number of records and memory cap. As input records, this script supports the following parameters: 1M_256, 5M_128, 10M_64, 100M_16, ALL.

But the MPI+FastFlow version has a different script due to different requirement. To run the simulation of MPI+FastFlow, use the following command:

```
$ cd "project-root"
$ cd src/
$ bash run_mpi.sh 32
```

3 Implementation

3.1 Sequential Version

As per the project description I have leveraged `std::sort()` function from `C++`. But to make performance comparison and suitable study of speedup and efficiency, we need a program that is purely sequential. For that I have programmed another version where reading, sorting and merging happens 100% sequentially. No communication, no overlapping and no overhead.

3.2 Tunable Constants

As we have discussed earlier about some Tunable constants which can help running the program in different machines in different scenarios.

3.2.1 DEGREE

Is a constant that depends on NUMBER OF WORKERS set to run in the program. Right now the decided value for this is:

$$\text{DEGREE} = WORKERS^2$$

This value is later used to determine the DISTRIBUTION_CAP. The intention is to decide a value(in bytes) for the emitter to read from the file in each turn and then slice them equally in such a way that one PE can put one slice easily in it's private cache.

3.2.2 DISTRIBUTION_CAP

Is the amount of bytes read from the input file in each turn. Right now the decided value is:

```
 uint64_t l1_cache = sysconf(_SC_LEVEL1_DCACHE_SIZE);
 uint64_t cap = l1_cache × DEGREE;
DISTRIBUTION_CAP = std::max(8388608UL, cap);
```

But a minimum of 8MiB has been set because much lower values creates communication overheads, and also very smaller writes in the `OOC` mode. But making this value larger will create an I/O delay and during that time workers will be idle. Also, we know that smaller read/writes are faster than big ones in mechanical discs. This constant is therefore tuned based on the current configuration of `spmcluster`, for this implementation.

3.2.3 MEMORY_CAP

Is the size-limit of MEMORY in GiBs. According to the project requirement, for each node we have a memory limit of 32GiB. But to test the program with a 32GiB input is hard and time consuming.So, instead of generating an input of 32GiB we can use this constant to limit the memory size and test the behavior of our program.

3.3 Algorithms

3.3.1 The Emitter

```
 1. Check file_size := get_file_size(input_file)
 2. IF file_size <= MEMORY_CAP THEN
     mode := IN_MEMORY
ELSE
     mode := OUT_OF_CORE
END IF
EMITTER():
 3.  segment_id := 0
 4.  WHILE NOT end_of_file DO
         segment := []
         accumulator := 0
         WHILE accumulator < DISTRIBUTION_CAP AND NOT end_of_file DO
             record := read_record(input_file)
```

```
            record_size := sizeof(key) + sizeof(len) + len
            IF accumulator + record_size > DISTRIBUTION_CAP AND segment NOT empty THEN
                BREAK
            END IF
            segment.append(record)
            accumulator := accumulator + record_size
        END WHILE
        ranges := slice_ranges(segment.size(), DEGREE)
        FOR each range [L, R] in ranges DO
            slice := segment[L:R]
            task := Task{slice, mode, segment_id, slice_index}
            ff_send_out(task)
        END FOR
        segment_id := segment_id + 1
    END WHILE
```

3.3.2 The Worker

```
WHILE TRUE DO
    task := receive_task()
    IF task is EOS THEN
        BREAK
    END IF
    std::sort(task.slice)
    result := TaskResult{task.slice, task.mode, task.segment_id, task.slice_index}
    IF task.mode = IN_MEMORY THEN
        result.kind := InMemBatch
    ELSE
        result.kind := SortedSlice
    END IF
    send_to_collector(result)
END WHILE
```

3.3.3 The Collector

```
COLLECTOR():
1.  segments_map := empty map[segment_id =: slices[]]
2.  inmem_batches := []
3.  run_paths := []
4.  WHILE receiving results DO
    result := receive_from_worker()
    IF result.kind = InMemBatch THEN
        inmem_batches.append(result.slice)
        ELSE IF result.kind = SortedSlice THEN
            segments_map[result.segment_id].append(result.slice)
            IF segments_map[result.segment_id].size() = DEGREE THEN
                run_path := k_way_merge_to_file(segments_map[result.segment_id])
                run_paths.append(run_path)
                delete segments_map[result.segment_id]
            END IF
        END IF
    END WHILE
5.  IF mode = IN_MEMORY THEN
    k_way_merge_to_file(inmem_batches, output_file)
    ELSE
        FOR each segment_id in segments_map DO
            IF segments_map[segment_id] NOT empty THEN
                run_path := k_way_merge_to_file(segments_map[segment_id])
                run_paths.append(run_path)
            END IF
        END FOR
        k_way_merge_runs(run_paths, output_file)
        delete_files(run_paths)
    END IF
```

The core algorithm for all versions is same. Based on the technological stack, a few additions are made but they didn't make any significant change.

3.4 FastFlow

The FastFlow implementation uses the farm pattern (building-blocks) with three stages: Emitter, Worker, and Collector. The program first checks if the input file size fits within MEMORY_CAP to decide between in-memory or out-of-core mode. The Emitter reads the input file and creates segments bounded by DISTRIBUTION_CAP, then divides each segment into DEGREE cache-sized slices using the slice_ranges function. These slices are distributed to workers using set_scheduling_ondemand(DEGREE), which enables automatic pull-based load balancing where idle workers request tasks from the emitter. FastFlow handles back-pressure automatically through internal bounded SPSC(Single Producer Single Consumer) channels.When all workers are busy and their input queues are full, `ff_send_out` blocks the emitter until workers consume tasks. Workers sort their assigned slices using std::sort and send results to the Collector.
In out-of-core mode, the Collector accumulates slices by segment_id and when DEGREE slices arrive for a segment, it immediately performs a k-way merge using a min-heap and writes the sorted segment(one DISTRIBUTION_CAP) to a temporary run file, then frees memory. After all segments are processed, incomplete segments are flushed and a final k-way merge of all run files produces the output. In in-memory mode, the Collector accumulates all sorted slices and performs a single k-way merge directly to the output file. **The implementation achieved similar results like OpenMP, but required significantly less code with no manual SafeQueue implementation, no mutex or condition_variable management, and no explicit synchronization primitives. The simplified code structure demonstrates that high-level structured parallelism frameworks can deliver competitive performance with substantially reduced implementation complexity. With targeted optimizations such as tuning DEGREE and DISTRIBUTION_CAP values, experimenting with custom FastFlow schedulers, implementing asynchronous disk I/O, the FastFlow version has strong potential to match or even exceed other framework's performance.**

3.5 OpenMP

The OpenMP implementation evolved through multiple versions (openmpv1̃5) before reaching the final optimized version in **openmp.cpp**. The architecture follows a three-stage farm pattern using `#pragma omp task`: Emitter, Workers and Coordinator. Unlike FastFlow's built-in abstractions, this implementation requires a manual `SafeQueue<T>` template class with explicit `std::mutex`, `std::condition_variable cv_consumer, cv_producer` for thread-safe communication between stages. The Emitter reads segments bounded by DISTRIBUTION_CAP, slices them into DEGREE cache-sized pieces, and pushes tasks to the task queue which blocks when full through `cv_producer.wait()` implementing explicit backpressure. Workers compete for tasks using `task_queue.pop()`, sort their slices with `std::sort`, and push results to the sorted queue. The Coordinator batches complete segments (DEGREE slices) and sends them to a write queue for out-of-core mode, or accumulates all data for in-memory direct writing. A dedicated writer thread pool of WORKERS/2 threads processes write tasks in parallel to overlap sorting with I/O operations (this is completely additional compared to FastFlow. I did nothing like this in FastFlow, yet FastFlow provides better results than OpenMP), requiring an additional `std::mutex paths_mutex` with `std::lock_guard` to protect the shared `run_paths` vector. Termination uses poison pills: the Emitter sends num_workers poison pills to workers, workers forward them to the Coordinator, and the Coordinator sends num_writers poison pills to writers. Queue sizes are manually tuned to `DISTRIBUTION_CAP * WORKERS` for task and sorted queues, and `num_writers` for the write queue to balance buffering and memory consumption. After the pipeline completes, a final k-way merge combines all run files into the output. This implementation achieved similar results like FastFlow, but required substantially more code—over 80 additional lines for SafeQueue implementation, explicit synchronization primitives throughout the farm, careful queue size tuning, and manual coordination of all inter-stage communication compared to FastFlow's declarative approach.

3.6 MPI+FastFlow

The MPI+FastFlow hybrid implementation extends the solution to a multi-node systems by combining MPI for inter-node communication with FastFlow for intra-node parallelism. I have used master-worker pattern where the master process (rank 0) reads the input file, creates segments bounded by `DISTRIBUTION_CAP`, and dynamically distributes them to idle worker nodes using `MPI_Send` with a custom `MessageHeader` protocol containing segment metadata and item counts. Each worker node (rank 1+) receives a segment via `MPI_Recv`, instantiates a complete FastFlow farm (Emitter, Workers, Collector) to process that segment internally, performs k-way merge of the sorted slices, and sends the merged result back to the master using `MPI_Send`. The master collects all sorted segments and performs a final k-way merge to produce the output, writing intermediate run files for out-of-core mode or merging directly for in-memory mode. I chose FastFlow over OpenMP for the hybrid version because FastFlow's encapsulated farm pattern is significantly easier to integrate with MPI where each worker node simply instantiates a self-contained farm without managing explicit `SafeQueue`, `mutex`, or `condition_variable` primitives that would require careful coordination with MPI communication. OpenMP's task-based parallelism with manual synchronization would introduce complexity in managing thread pools alongside MPI message passing, particularly handling back-pressure and termination across both frameworks. FastFlow's automatic internal synchronization through bounded SPSC channels operates independently of MPI, providing clean separation between inter-node (MPI) and intra-node (FastFlow) parallelism. Additionally, FastFlow farms can be created and destroyed per segment without coordination overhead, while OpenMP would require persistent thread pools and more complex state management across MPI operations. This architectural simplicity allows the hybrid implementation to scale horizontally across nodes while maintaining the same efficient intra-node parallelism as the single-node FastFlow version.

4 Performance Evaluation

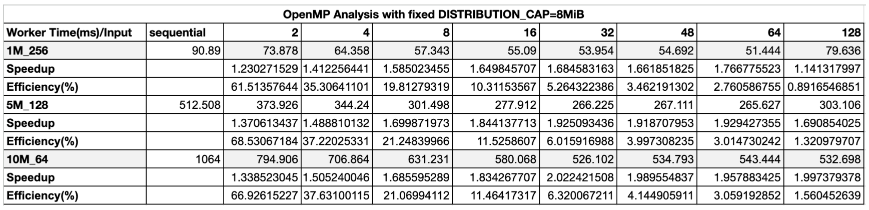
In this section we are going to discuss the performance analysis of our implementations. To achieve different results we have varied number of records with different payload size and workers. The tests are conducted in `spmcluster.unipi.it` with appropriate `SRUN` commands. For the tests we are considering 3 different inputs: `1M_256, 5M_128, 10M_64` and a set of workers: `2,4,8,16,32,48,64,128`. The goal is to check oversubscription issues, speedup and efficiency of the program between two technologies and single vs multi-node.

4.1 The Analysis

The timings are studied in different stages of the implementation. I have calculated individual timings for `Emitter, Worker and Collector`. The `Worker's` time is purely computational(sorting). And one final timing has been recorded for the overall execution of the program. So, during the performance analysis we will consider both `Worker's` and `Overall` time. We will analyze our program by varying `Number of Records(N), Payload(P) and Workers(W)`. As we have implemented `FARM`, the emitter, worker and collector, all of them are considered as FastFlow/OpenMP threads, hence workers. But in the implementation `Number of Workers(W)`(who performs the sorting) is equal to W. The reason I did that is because, Emitter and Collector are not always busy processing. While testing the implementation, I have noticed that the Collector is idle most of the time(specifically for IN_MEMORY operation). When the Collector is in action, the emitter and workers are idle. So, basically this is a trade-off situation and we can generate 2 extra threads(Emitter and Collector) to confirm how our program behaves with N number of pure workers.

4.1.1 Varied N, P and W

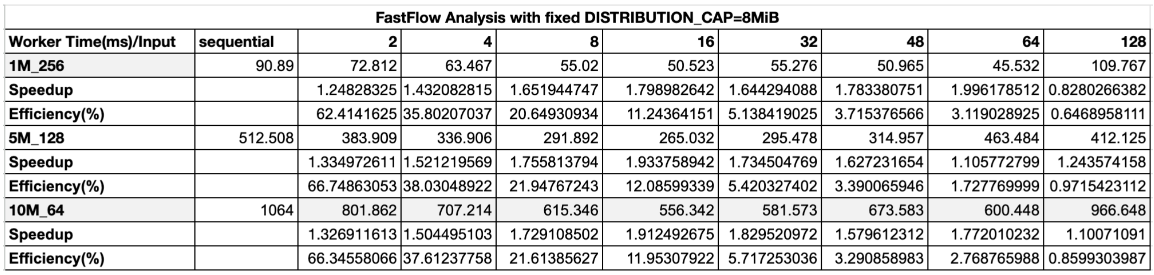
For number of records `N`, we are considering 1M(million), 5M and 10M with payload size 256, 128 and 64. For the testing I have also put a fixed value of 8MiB as the `DISTRIBUTION_CAP`, so that all setup read and process same amount of data.



OpenMP Analysis with fixed DISTRIBUTION_CAP=8MiB

Worker Time(ms)/Input	sequential	2	4	8	16	32	48	64	128
1M_256	90.89	73.878	64.358	57.343	55.09	53.954	54.692	51.444	79.636
Speedup		1.230271529	1.412256441	1.585023455	1.649845707	1.684583163	1.661851825	1.766775523	1.141317997
Efficiency(%)		61.51357644	35.30641101	19.81279319	10.31153567	5.264322386	3.462191302	2.760586755	0.8916546851
5M_128	512.508	373.926	344.24	301.498	277.912	266.225	267.111	265.627	303.106
Speedup		1.370613437	1.488810132	1.699871973	1.844137713	1.925093436	1.918707953	1.929427355	1.690854025
Efficiency(%)		68.53067184	37.22025331	21.24839966	11.5258607	6.015916988	3.997308235	3.014730242	1.320979707
10M_64	1064	794.906	706.864	631.231	580.068	526.102	534.793	543.444	532.698
Speedup		1.338523045	1.505240046	1.685595289	1.834267707	2.022421508	1.989554837	1.957883425	1.997379378
Efficiency(%)		66.92615227	37.63100115	21.06994112	11.46417317	6.320067211	4.144905911	3.059192852	1.560452639

Figure 1: OpenMP Working Time



FastFlow Analysis with fixed DISTRIBUTION_CAP=8MiB									
Worker Time(ms)/Input	**sequential**	**2**	**4**	**8**	**16**	**32**	**48**	**64**	**128**
1M_256	90.89	72.812	63.467	55.02	50.523	55.276	50.965	45.532	109.767
Speedup		1.24828325	1.432082815	1.651944747	1.798982642	1.644294088	1.783380751	1.996178512	0.8280266382
Efficiency(%)		62.4141625	35.80207037	20.64930934	11.24364151	5.138419025	3.715376566	3.119028925	0.6468958111
5M_128	512.508	383.909	336.906	291.892	265.032	295.478	314.957	463.484	412.125
Speedup		1.334972611	1.521219569	1.755813794	1.933758942	1.734504769	1.627231654	1.105772799	1.243574158
Efficiency(%)		66.74863053	38.03048922	21.94767243	12.08599339	5.420327402	3.390065946	1.727769999	0.9715423112
10M_64	1064	801.862	707.214	615.346	556.342	581.573	673.583	600.448	966.648
Speedup		1.326911613	1.504495103	1.729108502	1.912492675	1.829520972	1.579612312	1.772010232	1.10071091
Efficiency(%)		66.34558066	37.61237758	21.61385627	11.95307922	5.717253036	3.290858983	2.768765988	0.8599303987

Figure 2: FastFlow Working Time

Figures 1 and 2 we can clearly state that parallelism is not `proportional` to number of workers. We started having improved working time as number of workers increases, but after a certain point it starts decreasing again. This explains the issue of `oversubscription`. In our `spmcluster`, we have 20 physical cores with 2 threads in each core making it 40 logical cores. The machine also has 640KiB of data cache for each physical core with NUMA aware technology. From the table we can see that the program performs optimally with 32-64 workers depending on the workload, and the performance drastically degrades with 128 workers. This is clearly an oversubscription issue where the number of threads exceeds available logical cores, causing excessive context switching and cache thrashing.The FastFlow implementation shows exact same behavior but it beats the performance of OpenMP by a few milliseconds. Even though OpenMP beats FastFlow by a few milliseconds again in total execution time, with more optimization we can achieve similar or even better results.

The program behavior also changes based on Number of records and payload size. For each computation, there is a fixed cycle cost which increases with number of records. If we have small payload size, the workers can accumulate more records into its private memory(caches). Processing those records will have more fixed overhead than less items in the private memory(caches). Also processing more items increases possibility of higher cache misses. But it is not universal that with more records we will always get bad performance, it is always a trade-off.

4.1.2 Speedup and Efficiency

`Figure 2` illustrates that the speedup is not `super-linear` but `sub-linear`. It is not guaranteed that with more workers we will have better performance. We will achieve the optimal performance with optimal number of workers for each type of input.

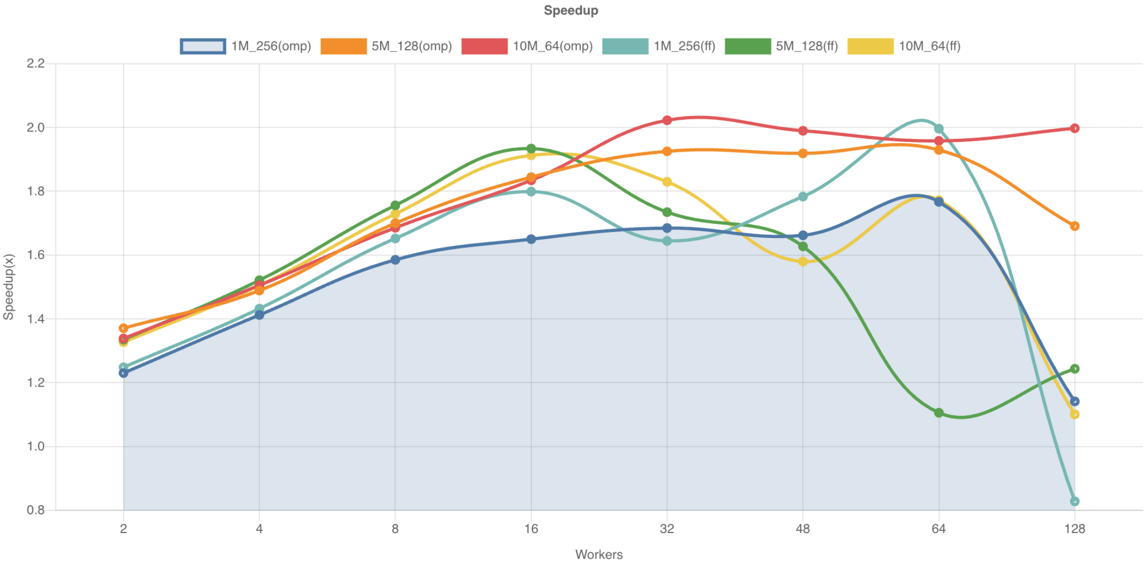


Figure 3: Speedup

By varying number of records and payload size we can see that for different inputs we have optimal speedup with different number of workers. For example: we achieved the highest speedup for 1M records of 256 bytes payload with 64 workers in OpenMP. Also with FastFlow we achieved highest speedup for 1M_256 with 64 workers. But for

5M records with 128 bytes payload the result changes significantly. We achieved highest speedup with 64 workers in OpenMP and only with 16 workers in FastFlow. The same is for 10 million records with 64 bytes payload, we achieved highest speedup with 32 workers in OpenMP and with 16 workers in FastFlow. This means FastFlow is more efficient for our implementation.

For the efficiency [`Figure 3`] we see a continuously degrading curve. This clearly demonstrates that achieving speedup with N workers does not guarantee efficient resource utilization. We may be paying a significant cost in additional workers for marginal performance improvements. The critical question becomes: what is the optimal parallelism degree ρ that balances speedup and efficiency? There is no universal answer, as the choice depends on optimization criteria and constraints. We select ρ based on the scenario:

- **Time-critical applications:** When deadline constraints dominate, choose ρ maximizing speedup regardless of efficiency (e.g., 64 workers for 1.80× speedup at 2.8% efficiency).
- **Resource-constrained scenarios:** When computational resources are limited or expensive, choose ρ where efficiency remains acceptable (typically $>$50%, or before the "knee" of the speedup curve where diminishing returns emerge). For our workload, this occurs around 8-16 workers.
- **Balanced approach:** Evaluate the marginal cost—if doubling workers yields $<$20% speedup improvement, the additional resources are poorly utilized and a lower ρ is preferable.

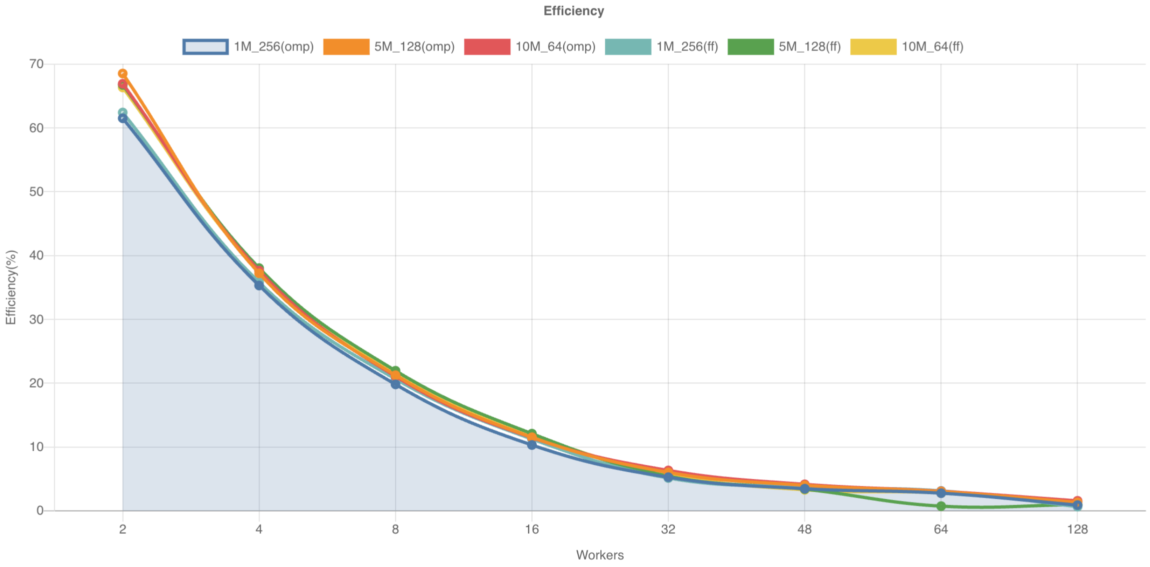


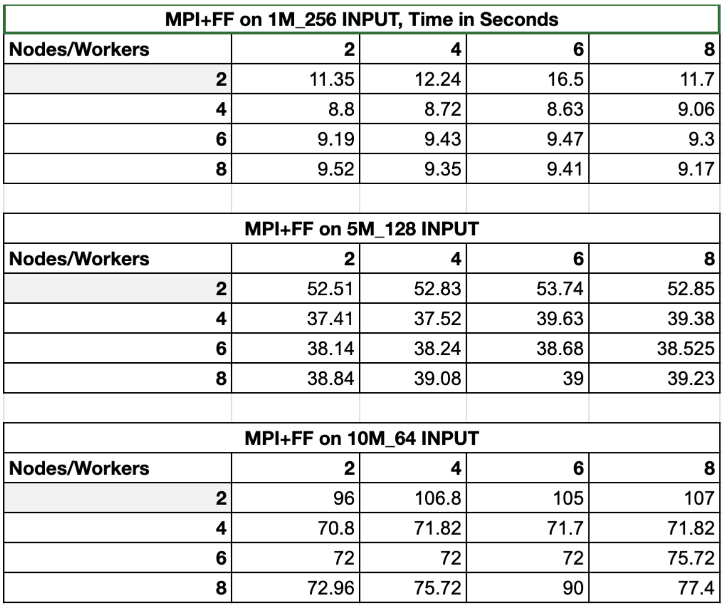
Figure 4: Efficiency

4.1.3 Strong vs Weak Scalability

For the weak and strong scalability we will consider $T^{farm}_{service}$ time for all executions. Till now we had our discussion on `Execution Time`(sorting-time) only to have a clear picture of workers' performance. But to discuss scalability we have to take `Emitter` and `Collector's` time into consideration as well. The service time also reflects a similar scenario where we can see the sub-linear behavior. The service time of `FARM` has been calculated by the this equation:

$$T^{farm}_{service} = max\{\frac{T^w_s}{K}, T^e_s, T^c_s\}$$

From the equation and table`[Figure 5]` data we can say that the `Collector Stage`(detailed in bottleneck discussion) dominates the performance, but there is a difference in varied nodes. According to `Strong Scalability` we should achieve reduced service time(super-linear speedup) on a fixed problem size with increasing number of `PEs`. Let us fix our test input as 10 million records with 64 bytes long payload`(10M_64)` with 8 workers. We can see the execution time is decreasing as we include more resources`(NODES)`. We start with 2 Nodes and 8 workers with a service-time of `107 seconds`, and gradually with `4, 6, 8` nodes the time reduces. But from 6^{th} node it starts increasing again. Which is not `super-linear` but `sub-linear`[Figure 6], and proves `Amdahl's Law` stating that



MPI+FF on 1M_256 INPUT, Time in Seconds				
Nodes/Workers	**2**	**4**	**6**	**8**
2	11.35	12.24	16.5	11.7
4	8.8	8.72	8.63	9.06
6	9.19	9.43	9.47	9.3
8	9.52	9.35	9.41	9.17

MPI+FF on 5M_128 INPUT				
Nodes/Workers	**2**	**4**	**6**	**8**
2	52.51	52.83	53.74	52.85
4	37.41	37.52	39.63	39.38
6	38.14	38.24	38.68	38.525
8	38.84	39.08	39	39.23

MPI+FF on 10M_64 INPUT				
Nodes/Workers	**2**	**4**	**6**	**8**
2	96	106.8	105	107
4	70.8	71.82	71.7	71.82
6	72	72	72	75.72
8	72.96	75.72	90	77.4

Figure 5: Service Time MPI+FastFlow(FARM)

performance will always be limited by the sequential part of the code. Since our program has dependencies on different memory levels like`Storage Devices, DRAMs, Caches` and `Concurrent Access on Data`, we can only scale a portion of the code to achieve optimal performance.

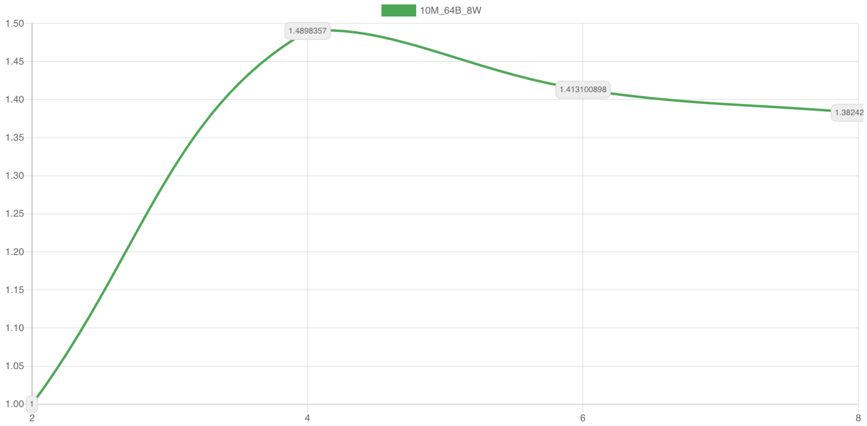


Figure 6: Strong Scalability Curve - MPI+FastFlow

If we consider f as the sequential part of the code, we can derive the equation from `Amdahl's Law`:

$$Speedup, S(\rho) = \frac{T_c(1)}{T_c(\rho)} = \frac{1}{f + \frac{1-f}{\rho}}$$

And for our program f is dominating, and this is why we achieve very little performance boost. The scenario is also similar for weak scalability `(Gustafson's Law)`. The law says if we scale the problem size along with the resources we can find more portion of the code to be benefitted by the parallelism. Let us consider α as the sequential part and β benefitted part from the parallelism. We still have f as the part which can be benefitted from the parallelism, and thus we can form this equation form the `Gustafson's law`:

$$S_\gamma(\rho) = \frac{\alpha f + \beta(1-f)}{\alpha f + \frac{\beta(1-f)}{\rho}}$$

From the table we can see the following behavior:

- **2 nodes:** 1M → 11.35s, 5M → 52.51s, 10M → 96s
- **4 nodes:** 1M → 8.8s, 5M → 37.41s, 10M → 70s

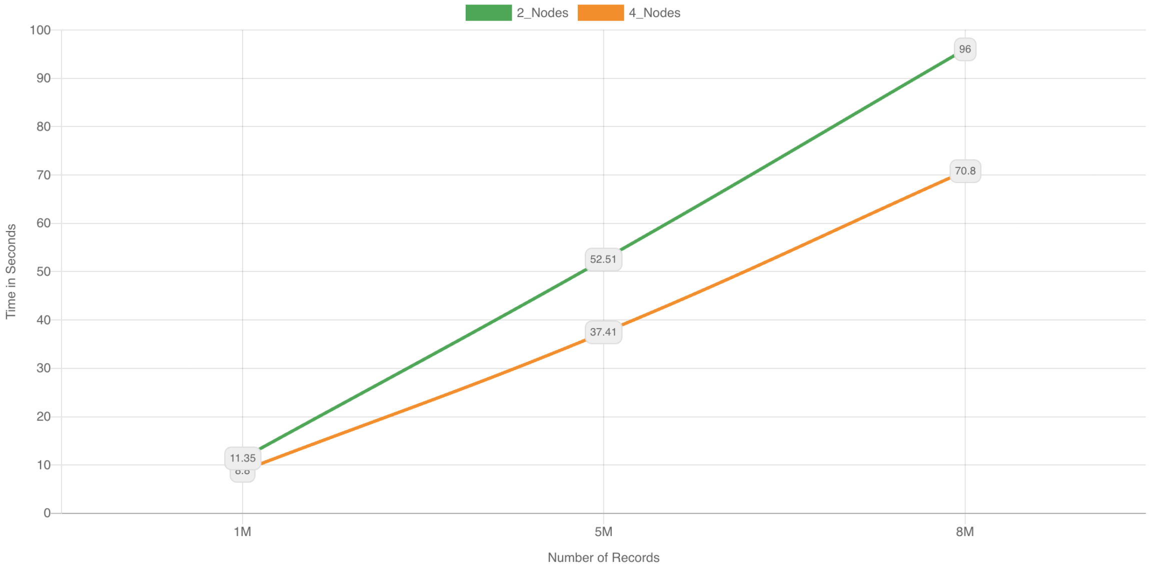


Figure 7: Weak Scalability Curve - MPI+FastFlow

5 Final Analysis

In this section I am going to discuss about the `cost-model` of the distributed solution I have proposed. Different time studies will be considered for the cost-model. To keep things simple I am not attaching the logs from the code, but they can be enabled in the `farm.cpp` codebase and studied.

5.1 Cost Model

Lets denote the parameters for our cost-model first `[for 1M records with 256 bytes payload and 1 worker]`:

Inter-arrival Time, $= T_{arrival} = 29[ms]$

Service time of a Worker, $= T_s^w = 7[ms]$

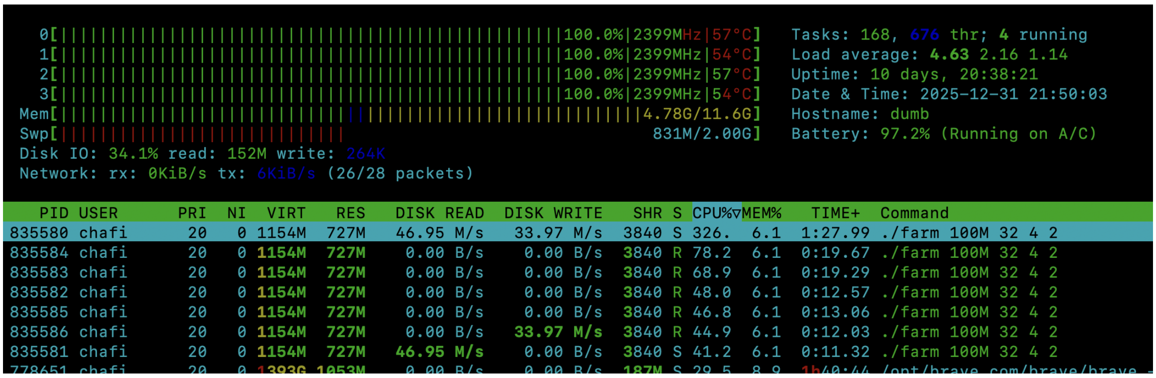
Emitter Total, $= T_s^e = 491[ms]$

Collector Total, $= T_s^c = 951[ms]$

Number of Optimal Workers, $K_{opt} = \dfrac{T_s^w}{T_{arrival}} = \dfrac{29}{7} = 4.1 \approx 4$

$$T_{service}^{farm} = max\{\frac{T_s^w}{K}, T_s^e, T_s^c\} = max\{4, 491, 951\} = 951[ms]$$

This is the `Service-Time` of our `FARM` implementation using FastFlow. Note that this service-time ignores the communication overhead but include communication over computation overlapping. We are emitting each segment equal to `DISTRIBUTION_CAP` bytes. From the second segment we are overlapping the communication between emitter and workers. But the significant part is the final merge, where we are doing k-way merge but still writing one big `<.bin>` file. That is the largest sequential part of our code which dominates everything(`strong-scaling by Amdahl's law`). But the program gives satisfactory results in the `MEMORY-OOC` mode where we are writing each small segments before the final merge. At that mode we achieve 2 ways `computation-to-communication` overlapping, because from the second segment we start overlapping emitting and sorting and from the third segment we achieve emitting-sorting-writing. And using that mode we can sort `TeraBytes`. Even though in this report I have discussed mostly the `IN_MEMORY` operation but in the implementation I have focused more on the `OOC` mode. The report pages are limited that's why I couldn't bring it here but I would request you to visit the github page if anyone is interested. Here is an image from a test of 100M records with 32 bytes long payload using FastFlow(farm).



```
   0[||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%|2399MHz|57°C]   Tasks: 168, 676 thr; 4 running
   1[||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%|2399MHz|54°C]   Load average: 4.63 2.16 1.14
   2[||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%|2399MHz|57°C]   Uptime: 10 days, 20:38:21
   3[||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%|2399MHz|54°C]   Date & Time: 2025-12-31 21:50:03
 Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||4.78G/11.6G]   Hostname: dumb
 Swp[|||||||||||||||||||||||||||||                              831M/2.00G]   Battery: 97.2% (Running on A/C)
 Disk IO: 34.1% read: 152M write: 264K
 Network: rx: 0KiB/s tx: 6KiB/s (26/28 packets)

  PID USER       PRI  NI  VIRT   RES   DISK READ  DISK WRITE   SHR S CPU%▽MEM%   TIME+  Command
835580 chafi      20   0 1154M  727M  46.95 M/s   33.97 M/s  3840 S 326.   6.1  1:27.99 ./farm 100M 32 4 2
835584 chafi      20   0 1154M  727M   0.00 B/s    0.00 B/s  3840 R 78.2   6.1  0:19.67 ./farm 100M 32 4 2
835583 chafi      20   0 1154M  727M   0.00 B/s    0.00 B/s  3840 R 68.9   6.1  0:19.29 ./farm 100M 32 4 2
835582 chafi      20   0 1154M  727M   0.00 B/s    0.00 B/s  3840 R 48.0   6.1  0:12.57 ./farm 100M 32 4 2
835585 chafi      20   0 1154M  727M   0.00 B/s    0.00 B/s  3840 R 46.8   6.1  0:13.06 ./farm 100M 32 4 2
835586 chafi      20   0 1154M  727M   0.00 B/s   33.97 M/s  3840 R 44.9   6.1  0:12.03 ./farm 100M 32 4 2
835581 chafi      20   0 1154M  727M  46.95 M/s    0.00 B/s  3840 S 41.2   6.1  0:11.32 ./farm 100M 32 4 2
778651 chafi      20   0 1393G 1053M   0.00 B/s    0.00 B/s  187M S 29.5   8.9 1h40:44 /opt/brave.com/brave/brave -
```

Figure 8: Memory Out of Core operation on 3.6GiB file with 1GiB Memory

`Figure [8]` demonstrates that we can sort large files in `OOC` mode without paralyzing the whole system. Even though the memory cap is 1GiB, the program is consuming more than 1GiB. It is very common because there are overheads through out the processing. A breif study is given under `/src/discussions/` with test codes to prove the point.

5.2 The Bottleneck Phase

This is the most important part of the whole report. By now we must have become hopeful to see a result that will amaze us! But it is not going to do so. Two main laws of parallel paradigm `Amdahl's and Gustafson's` law didn't just show the handicap of parallelism they actually broke the dilemma. That is why years after years computer scientists are still doing research on it. Parallelism is not just about `PEs` with higher frequency clock, it is a combination of the whole eco system. And in my program I can see that very clearly. Lets see two output of a run with 1M records of 256 bytes payload and 2, 4 and 8 workers.

- M: IN-MEMORY | R: 1M | PS: 256 | W: 2 | DC:8MiB | ET: 456.331 ms | WT: 98.008 ms | CT: 767.576 ms | TT: 1.277 s
- M: IN-MEMORY | R: 1M | PS: 256 | W: 4 | DC:8MiB | ET: 901.708 ms | WT: 96.079 ms | CT: 988.654 ms | TT: 1.944 s
- M: IN-MEMORY | R: 1M | PS: 256 | W: 8 | DC:8MiB | ET: 1.484 s | WT: 113.282 ms | CT: 874.681 ms | TT: 2.436 s

What do we see from the outputs? We can see that a significant cost is being paid at emitter and collector stage. While the workers are sorting the data much faster, emitter cannot read that fast from the disc and collector cannot write on the disc. Mechanical discs has limitations, even though these days we have SSDs and NVMEs but their speed is nothing comparable to the CPU. Also the cache-memory should be taken under consideration because CPUs recursively fetch data from the memory and operates. So, this is a minimum 3 way operation(ignoring other memory hierarchy): Disc → Memory → Cache. So, we can declare that for us in this problem `EMITTER & COLLECTOR` are the bottleneck phases. Now the question is, how can we get rid of it? Does this mean in all parallel programs we will have these same issues? The answer is **NO!**. Adopted solution always depends on the type of problem we have. **In this program we are not doing anything that is computationally heavy. We are reading binary bits from the file, forming items, sending to the workers and the workers are just performing CAS(compare & swap). Also, we introduced a technique for the workers so that they can sort everything privately. Also there is no data dependencies like reusing generated results from other workers. That is why workers became the fastest throughout the whole operation.** So, we could have simply parallelized the `EMITTER & COLLECTOR`, isn't it? Yes! That could be one solution which I have also tried but it introduces another problem. `Race-Condition` is another significant term in parallelism. The goal of our program is to sort the data, which means the order must be maintained. Now, the workers are sorting the data privately without knowing which slices of the input are sorted by other workers. The final `k-way` merge is performed by the collector who ensures the sorted manner of the records. Imagine having multiple collectors trying to write the records in the same binary file. **It is common that they will write intermediate bits which will not only disturb the sorting nature but also corrupting the information. To solve that problem we can introduce locks but that doesn't solve the problem because multiple threads of the collector will fight to achieve that lock which will generate more communication overheads.** So, I will say that this solution is not 100% optimal and has so many opportunities for improvement in future. But the `out-of-core` mode is really helpful for machines with limited resources. **Also for shared hosting machines where we have multiple users accessing the database frequently**, the ooc mode is really helpful for slicing big files into smaller ones for better efficiency. That is all, but lets discuss some other problems which were also crucial for the better performance.

5.3 Problems & Adopted Solutions

During the development I have faced several problems regarding data-structure, load-distribution, synchronization and I/O handling. Each of these problems have been solved based on the theoretical part we are taught during the course. Even though I cannot say I have solved 100% of them, but I can assure you this is a good starting point. From this point it can be more optimized. Lets discuss the problems and solutions.

5.3.1 Data Structure

In the initial implementation I used `std::vector<uint8_t>` for storing variable-length payloads, which created a critical memory consumption issue. The structure `std::vector<Item>` contained items where each Item had a `std::vector<> payload`, resulting in nested vectors. The fundamental problem is that `std::vector<uint8_t>` consumes 24 bytes of overhead per instance to store internal pointers (begin, end, capacity) and bookkeeping information, while a simple `uint8_t*` pointer consumes only 8 bytes. For large datasets, this overhead becomes catastrophic. Experimental measurements[**] show that 100 million records using `std::vector` consumed 2288 MB of memory compared to only 762 MB using the `CompactPayload` pointer-based approach, a difference of 1526 MB (66% overhead). This represents a waste of 16 bytes per record purely for vector metadata. For the 100M dataset, this overhead alone required an additional 1.5 GB of memory before storing any actual payload data. The `CompactPayload` class was designed to replace the vector by using a single `uint8_t*` pointer that points to a contiguous memory block containing the payload size (uint32_t) followed by the actual data bytes. This design maintains the flexibility of variable-length payloads while reducing per-record overhead from 24 bytes to 8 bytes. The 16-byte savings per record prevented memory exhaustion and system crashes during sorting operations on large datasets, demonstrating that careful data structure selection is critical for memory-bounded external sorting algorithms.

5.3.2 OpenMP

1. **Sequential Bottleneck Due to No Data Distribution:** The first version has a critical design flaw where the emitter reads the entire `MEMORY_CAP` amount of data before workers could start processing. This meant parallelism only occurred during the sorting of slices, not during data reading. Workers remained completely idle while the emitter was reading, which defeated the purpose of parallel processing(computation overlapping communication). This sequential bottleneck was particularly severe for large datasets, as the emitter had to read gigabytes of data before any computation could begin.

   Solution: In version 3, we introduced the `DISTRIBUTION_CAP` parameter to limit how much data the emitter reads at once, forcing data distribution in smaller chunks and enabling workers to start processing while more data is being read.

2. **Memory Paralyzed and System Crashes:** Testing revealed a shocking memory consumption issue: a 2.7 GiB input file consumed 5.9 GiB of memory after reading. The root causes were struct overhead (16 bytes per Item), memory allocation metadata (8-12 bytes), and vector capacity growth (allocating in powers of 2). For 100 million records with 128-byte payloads, this overhead could exhaust system memory and crash the machine. Additionally, vectors were not freed until program completion, and the emitter would start reading the next segment before workers finished processing the current one, compounding the memory pressure.

[**] **the experimental program can be found under /src/discussion/tests**

Solution: Version 2 introduced memory cleanup by freeing memory after each intermediate write, preventing system paralysis. Version 3 further mitigated this by using `DISTRIBUTION_CAP` to limit simultaneous memory allocation. The vector overhead itself was later addressed with the `CompactPayload` data structure.

3. **No Pipeline Parallelism:** In the out-of-core scenario, after the emitter read `MEMORY_CAP` worth of data and workers sorted it, the collector would perform the k-way merge and write the segment while the emitter and all workers sat completely idle. For processing 20 GiB of data with a 2 GiB `MEMORY_CAP`, this meant 10 sequential cycles of read-sort-merge-write with no overlap, resulting in disastrous performance. The stages were not pipelined, so only one stage (reading, sorting, or merging) was active at any time.

   Solution: Version 3 achieved partial pipeline parallelism by introducing `DISTRIBUTION_CAP`, though the emitter would pause during processing and workers would pause during the next read. Version 4 improved this significantly by introducing queues for unsorted and sorted tasks, allowing the emitter, workers, and collector to operate more concurrently.

4. **Static Scheduling and Load Imbalance:** Even with `DISTRIBUTION_CAP` in version 3, workers became idle because the number of tasks equaled the number of workers, and the scheduling pragma (schedule(dynamic)) was ineffective. When workers finished their slices at different times due to variable payload sizes, faster workers had nothing to do while slower workers were still processing. This under-utilization was exacerbated by the emitter pausing while workers processed, creating alternating periods of reading and computing instead of continuous overlap.

   Solution: Version 4 introduced task queues with mutex locks and generated `workers * n` tasks instead of just tasks equal to workers, providing more fine-grained work distribution. This kept workers busy with smaller tasks and improved load balancing across threads.

5. **Unbounded Queue Growth and Memory Pressure:** Version 4 introduced queues without size limit. The sorted task queue would grow unboundedly because the collector was slower at merging and writing than workers were at sorting. Memory was not released immediately after sorting, causing accumulation. Additionally, writing intermediate files of `MEMORY_CAP` size was slow, and during these slow writes, the emitter continued reading and workers continued sorting, filling the queues endlessly. This led to IO contention and risked memory exhaustion despite solving earlier distribution issues.

   Solution: Version 5 introduced bounded queues with explicit queue size limits, implementing back-pressure that would pause the emitter when queues filled up. This prevented unbounded memory growth and ensured the system operated within memory constraints, especially beneficial for **out-of-core operations**.

6. **Collector Bottleneck and Suboptimal Cache Utilization:** Later I found the collector as the bottleneck, unable to keep up with the rate at which workers produced sorted slices. Additionally, earlier versions did not consider cache hierarchy when sizing work units. Slices were not optimized for L1 cache size, meaning workers performed sorts on data larger than their fastest cache level, incurring memory access penalties. The intermediate write strategy also needed optimization, as a single collector writing sequentially couldn't match the throughput of multiple parallel workers.

   Solution: The final version introduced multiple optimizations: `DEGREE` parameter to create slices that fit in L1 cache (improving sort performance), `DISTRIBUTION_CAP` sized as `L1_CACHE * DEGREE` to balance memory usage and cache efficiency, and `WRITERS = WORKERS/2` threads dedicated to parallel intermediate writes from the sorted task queue. This allowed the emitter and workers to continue operating while a pool of writer threads handled I/O, eliminating the collector bottleneck.

5.3.3 FastFlow

In FastFlow I didn't face these much issues because most of the functionalities were taken care of by FastFlow. The memory overflow and no-pipeline parallelism issues were there but when I fixed them for OpenMP, it also got fixed for FastFlow. Because that was the problem of data-parallelism not of FastFlow. The automated scheduling of FastFlow made things much easier. `And that is also the reason why I preferred FastFlow over OpenMP for the MPI implementation`. MPI itself introduces communication overheads, and to handle things manually with openmp would make it more difficult.

6 Conclusion

I would like to say some final words before concluding. Even though it is a student project but still this project covers almost every concept of parallel and distributed systems. It is a good starting point to master someone's skill in parallel programming. We got introduced with technologies like FastFlow, OpenMP and MPI through this project. If we combine our theoretical knowledge with these technologies we can design efficient programs to solve serious problems. Even in my thesis I have planned to use the same concepts for path computation and spectrum allocation. I have said that earlier and I am going to say it again, this project is not 100% optimized but it is a good starting point. It was my course final project but it has topics worthy enough of having a thesis work. Points like data distribution, synchronization with cache levels, sequential or parallel access to single entity and optimization for legacy systems can bring new ideas. Finally, I would like to thank our professor for introducing FastFlow. The framework eases the work for a programmer by handling so many things automatically. By studying the framework more, we can come up with different solutions of different problems.

References

[1] Course Contents of Parallel and Distributed Systems: Paradigms and Models [2024/2025]

[2] FastFlow Parallel Framework [http://calvados.di.unipi.it/][https://github.com/fastflow/fastflow]

[3] OpenMP [https://www.openmp.org/resources/openmp-compilers-tools/#compilers]

[4] MPI [https://www.open-mpi.org/]