SPM Project: Distributed External Memory Sorting

Marco Pampaloni
Department of Computer Science
m.pampaloni2@studenti.unipi.it
June 30, 2025

Contents

1. Introduction

1.1. Problem Statement

The project specification required implementing a scalable parallel external memory merge sort. The file to sort is a sequence of records composed by a 64 bit key, which is used for sorting, and a variable-sized payload with arbitrary byte length $8 < \text{len} < \text{PAYLOAD\_MAX}$. The file cannot be assumed to entirely fit in a single's node central memory.

2. Implementation

2.1. Architecture

The strict requirements outlined above prevent a naive merge-sort implementation, forcing a smart use of heap memory allocations and partial results collection. In particular, since the file cannot fit in internal memory, I decided to employ a classical external-memory merge sort approach, given its I/O optimality guarantees on shared-memory systems. This algorithm, however, poses many implementation challenges that are not easily surmountable in practice.

The chosen algorithm works on two phases:

- **Phase 1**: The input file is read sequentially from disk. This incurs an optimal number of I/O-heavy system calls. The unsorted records are deserialized and collected in large batches that can fit in internal memory. Each batch is sorted using a fast in-memory sorting algorithm and subsequently stored on disk as a parially sorted collection of records.
- **Phase 2**: After the input file is exhausted and, *critically*, after each batch has been sorted and written to disk, a final merging phase can be executed; each locally sorted file is read sequentially while keeping a single block of data in memory for each file to minimize random I/O operations. At each merging iteration, the single best record (according to its comparison key) is chosen and written in a write buffer. When the buffer gets full it can be permanently stored on disk. This selection operation can be done in logarithmic time using a min-heap. After each partial file has been consumed, the algorithm terminates and the entire input file has been sorted and stored on disk.

Both phases are I/O-bound, but the second one in particular is inherently sequential and cannot be parallelized. Moreover, it has to wait for the first phase to finish so stream-parallelism approaches are limited.

2.2. Hardware

Local development and initial tests for shared-memory implementations have been carried out on an AMD Ryzen 7950X machine with 16 cores and 32 threads, while final tests have been performed on the University's *spmcluster*. Even with a fast PCIE Gen 4.0 Nvme drive with 7GB/s sequential read bandwidth, the implementation remained I/O-bound, though more data parallelism could be employed during the sorting phase.

2.3. Shared-Memory Algorithm

To mitigate some of the performance issues caused by the low disk-bandwidth of the *spmcluster*, I decided to adopt the following optimizations:

- The implementation features buffered I/O to efficiently read chunks of binary data from the input files. This reduces heavy read system calls;
- Given the variable nature of the records' sizes subject to sorting, allocating the memory necessary for serializing/deserializing the records' payloads on the stack cannot be done. Instead each payload has to be allocated on the heap, which should be done repeatedly and with high-frequence during both phases. For this reason, I devised a simple block-contiguous Memory Arena allocator which allocates large chunks of free-store memory upon creation and returns zero-copy contiguous views of such memory when a new

allocation of given size is requested. This greatly reduces memory allocations and provides a flexible way of automatically managing allocations/deallocations using batches of records wrapped in smart pointers.
- Both phases employ stream-parallelism to hide I/O operations behind computation.
- Async I/O by means of double buffering and C++'s threads has also been tried, but did not improve performance (this was perhaps due to the overhead of thread management outweighing the benefits on this particular system) and I thus decided to not use it to unclutter the implementation.
- During Phase 1, batches are read sequentially and records are collected in contiguous regions of memory to exploit data locality. Meanwhile, different batches are allocated on distinct memory arenas to reduce false-sharing in later stages of the pipeline. This also improves memory utilization, since once a batch has been succesfully processed, all the memory of its memory arena can be freed. This is done automatically used smart pointers.

The following structs can be used to efficiently collect batches of records:

```
struct RecordView {
  uint64_t key;
  std::span<char> payload;

  auto inline operator==(const RecordView& other) const -> bool {
    return key == other.key;
  }
  auto inline operator<=>(const RecordView& other) const -> std::strong_ordering {
    return key <=> other.key;
  }
};

struct ArenaBatch {
  MemoryArena<char> arena;
  std::vector<RecordView> records;

  ArenaBatch(size_t batch_size, size_t arena_size) : arena(arena_size) {
    records.reserve(batch_size);
  }

  auto totalBytes(size_t header_size = sizeof(uint64_t) + sizeof(uint32_t)) -> size_t {
    return arena.used() + records.size() * header_size;
  }
};
```

The I/O business logic is wrapped in an utility object which is used throughout the implementation:

```
template <
    size_t BufferSize,
    typename Allocator = DefaultHeapAllocator<char>,
    typename RecordType = files::Record>
class BufferedRecordLoader;
```

By default, `BufferedRecordLoader` uses a `DefaultHeapAllocator` which simply allocates new contiguous blocks of memory whenever requested, but can also be provided with a `MemoryArenaAllocator` to employ the optimization described above. When this is the case, a `RecordView` has to be used as `RecordType`, which holds a non-owning zero-copy view over the contiguously allocated region of memory.

The sections below describe how the sorting algorithm has been implemented exploiting FastFlow's and OpenMP data and stream parallelism for shared-memory systems. A later section describes how the distributed merge sort has been implemented using MPI and OpenMP.

2.3.1. FastFlow

FastFlow's implementation closely follows the algorithmic description presented above, but employs both stream and data-parallelism to accelerate sorting and hide I/O latencies. In particular, it implements two pipelines executed back to back. The first one features an `Emitter` node that reads and collects batches of records from the input file and asynchronously sends them to a `BatchSorter` farm which locally sorts the records and sends them to a collector node that writes them on disk. On machines with very fast I/O like the one used for local development, this later stage can profitably be turned into a farm. However, tests performed on the *spmcluster* showed that this did not bring any benefit to the implementation and I thus chose to use a single worker (while keeping the implementation general with a `--num_writers` command line argument).

Finally, the last stage of the first pipeline collects the paths for the temporary sorted runs created by the previous stage for later reuse.

The second phase is implemented as a simple 2-stage pipeline that implements the merging strategy described earlier, with I/O latencies minimized by collecting small batches of sorted records from the merging phase, which is performed at the same time. Here I could not use the memory arena optimization because records within a batch could not necessarily be allocated from a single memory pool, due to the complex runs' intertwined relative ordering. Between the first and second pipeline, an implicit barrier is employed.

2.3.2. OpenMP

OpenMP's implementation is identical to FastFlow's one, in that it still basically defines two pipelines with a synchronization point in-between, but it does so leveraging OpenMP's task directives: once a batch of unsorted records is collected from the first stage, a new task is spawned that sorts and stores the records on disk. This approach perfectly scales with the speed of the underlying I/O device because it does not introduce any dependency between different tasks, which can spawn at the speed with which batches are collected and start sorting them in parallel. With fast disks or distributed systems this should work better than FastFlow's fixed farms of workers.

The second phase is again implemented as a simple 2-stage pipeline, but since we cannot rely on FastFlow's internal queues, I implemented a simple mutex-based thread-safe queue implementation to exchange batches of sorted records to the next and final stage. This is again implemented using a producer-consumer task setting:

```
#pragma omp parallel
  {
#pragma omp single  // Producer task, executed by a single thread
    {
#pragma omp task shared(out_file, task_queue)  // Consumer task
      {
        while (true) {
          auto batch_to_write = task_queue.pop();
          if (batch_to_write == nullptr)
            break;
          writeOutputBatch(std::move(batch_to_write), out_file);
        }
      } // consumer
      ...
#pragma omp taskwait
    } // single
  } // parallel
```

2.4. Distributed Algorithm

2.4.1. MPI + OpenMP

The distributed implementation of the external-memory merge sort algorithms leverages MPI for inter-node communication and OpenMP for intra-node parallelism. I chose to use OpenMP for its simplicity and because I found it to scale better with respect to the underlying hardware than a naive FastFlow's implementation. For this reason, most of the code from OpenMP's shared-memory implementation could be reused, wrapped into a `ParallelSorterOMP` class.

The shared-memory algorithm can be moved to a distributed setting with minimal modifications. The nodes network is divided into a single emitter (root) node responsible for reading the input file and collecting large batches of records, which is scattered to multiple worker nodes that independently employ the same shared-memory algorithm described earlier.

Because each record can hold a variable amount of bytes as payload, a 2-stage communication must be employed to send the data over the worker nodes (alternatively, a custom MPI data type that describes the memory configuration of the record batches could have been used, but it didn't suit well with the block-contiguous memory arena allocator employed). This involves performing two MPI calls:

- `MPI_Scatter(send_counts.data(), 1, MPI_INT, MPI_IN_PLACE, 1, MPI_INT, ROOT_RANK, communicator)`
- ```
  MPI_Scatterv(
      serialized_batch.bytes.data(),
      send_counts.data(),
      displacements.data(),
      MPI_BYTE,
      MPI_IN_PLACE,
      0,
      MPI_BYTE,
      ROOT_RANK,
      communicator
  );
  ```

The first one scatters the sizes in bytes of the buffers that each worker should allocate to receive the batch of record. The second function call sends the actual data, serialized in a byte stream, with the displacement array delimiting each batch within it. The workers receive the scattered counts, allocate buffers to accomodate the data and deserialize it using the same memory-arena-backed `BufferedRecordLoader` as the shared-memory implementation. Then a new task is spawned to sort the data and write it to disk, trying to hide I/O and communication latencies.

Finally, when the input file has been exhausted by the emitter node, an `MPI_Scatter` call with negative counts is performed, which signals the workers the EOF has been reached. At this point, each worker node can perform a local merge of each sorted run stored on disk, resulting in a single sorted file per node.

Given the NFS mounted file system available on the *spmcluster*, I decided to employ a fixed, rank-based, naming scheme for each node's last locally merged sorted file, so that no additional communication between nodes is necessary. The emitter node, then acts as a collector and eventually performs a final merge step using the sorted files generated by the workers.

3. Results

3.1. Experimental Setup

All performance evaluations were conducted using a synthetically generated base input file consisting of one million unsorted records. Each record was created with a randomly generated 64-bit key, while its payload size was chosen from a uniform distribution ranging between 8 and 64 bytes. For the weak scalability
  ```

analysis, this base file was scaled proportionally with the number of processes. A larger initial dataset was avoided to ensure the execution time of each experiment remained within the 10-minute limit imposed by the project requirements.

3.2. Analysis Limitations

The performance evaluation has been carried out on the *spmcluster*, whose hardware architecture poses significant challenges for collecting reliable and reproducible measurements: concurrent I/O-heavy tasks from other users on any node inevitably saturate the disk and network bandwidth of the single frontend, causing notable fluctuations in the measured execution times.

3.3. Performance Evaluation

As already pointed out, the configuraion of the cluster used for testing the distributed implementation of the algorithm presents a heavy I/O bottleneck that prevents proper scaling. In fact, the provided implementation only shows a very modest speedup for the single-node shared-memory implementation, while showing virtually no performance improvements for the distributed version. This section provides scaling, speedup and efficiency plots for each algorithm configuration.

3.3.1. OpenMP Shared-Memory Implementation

The first analysis focuses on the shared-memory implementation using OpenMP on a single node. As shown by the speedup curve in Figure 1, the algorithm benefits from multiple cores on CPU-bound tasks, but the gains quickly diminish.

The parallel efficiency, plotted in Figure 2, drops accordingly as more threads are added. This confirms that even on a single machine, performance is ultimately capped by the I/O throughput to the centralized NFS storage, preventing linear scalability.

3.3.2. FastFlow Implementation

A similar trend is observed for the FastFlow implementation. The stream-parallel pipeline model proves effective at hiding some I/O latency, leading to modest but consistent speedup as seen in Figure 3.

However, as the corresponding efficiency plot (see Figure 4) confirms, this approach also saturates quickly. The program cannot perform faster than the rate at which the underlying storage can provide data.

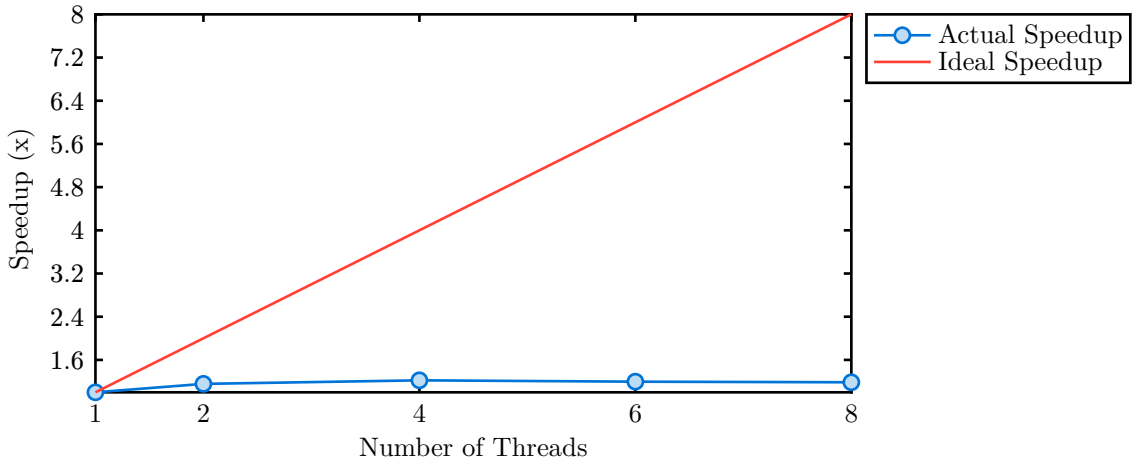

Figure 1: Parallel speedup of the OpenMP implementation. The curve shows sub-linear speedup, flattening out after 4 threads, which indicates that the algorithm is becoming limited by bottlenecks other than CPU, likely I/O bandwidth to the NFS server.

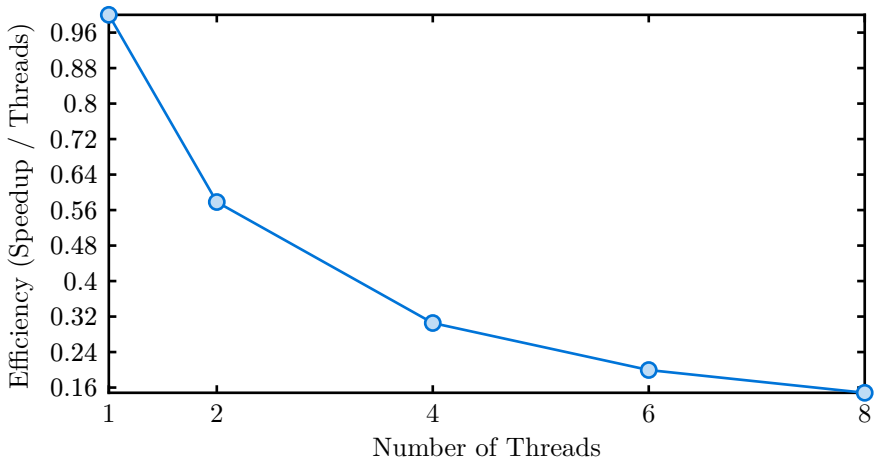

Figure 2: Parallel efficiency of the OpenMP implementation, calculated as Speedup / Threads. The curve shows a clear decline as more computational resources are added.

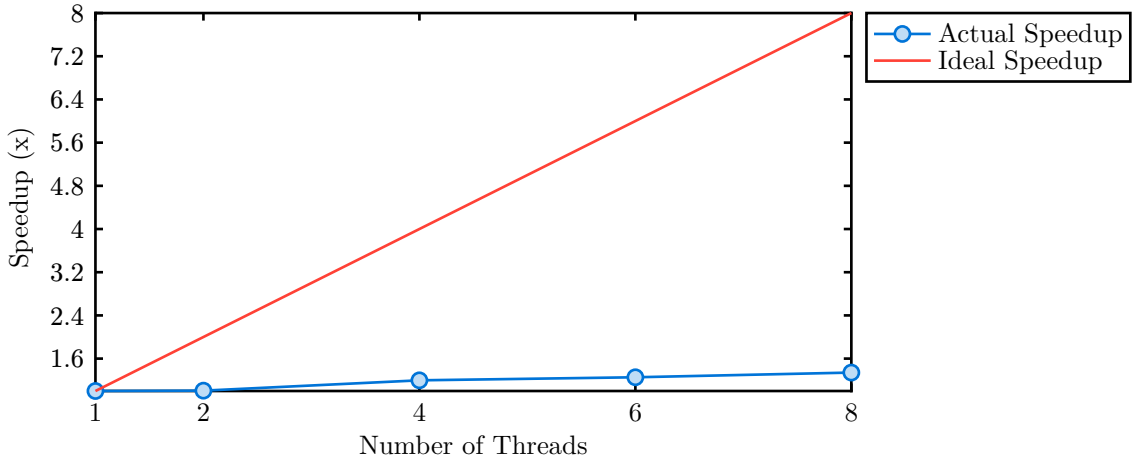

Figure 3: Parallel speedup of the FastFlow implementation. The speedup is modest, confirming that while the pipeline architecture is effective, its scalability is ultimately capped by the performance of the shared I/O subsystem.

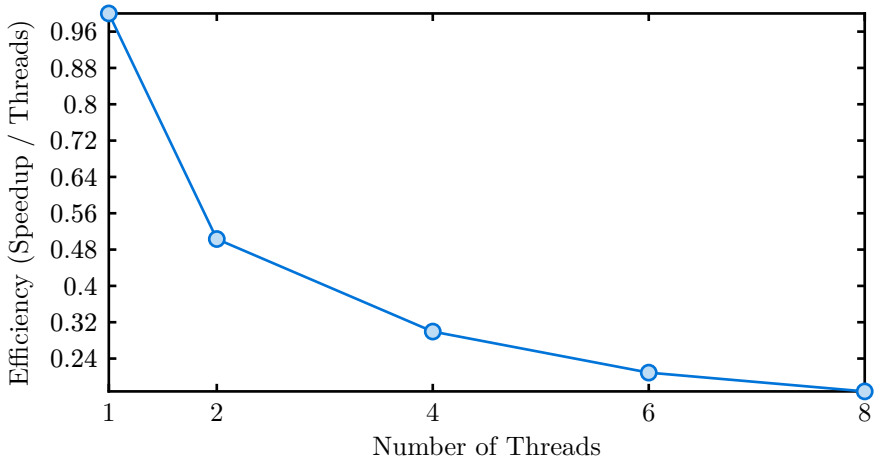

Figure 4: Parallel efficiency of the FastFlow pipeline implementation.

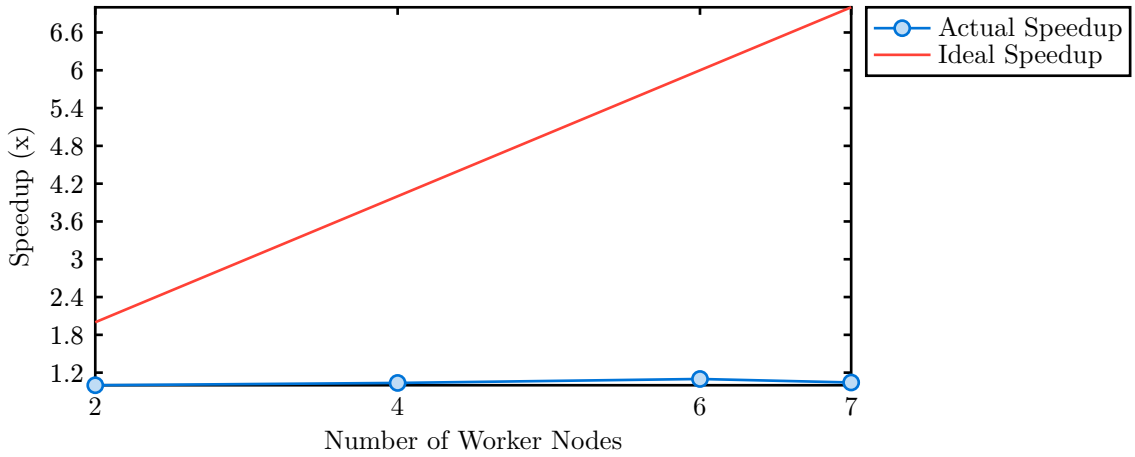

Figure 5: Distributed speedup (strong scalability) of the MPI implementation, relative to the 2-node baseline. The curve is nearly flat, confirming that the bottleneck created by the centralized NFS storage and 10GbE network prevents the algorithm from scaling effectively as more nodes are added.

3.3.3. MPI Distributed Implementation

Finally, scalability is plotted for the distributed MPI implementation. The strong scalability test (Figure 5) shows a nearly flat speedup curve, indicating that adding more nodes provides almost no performance benefit for a fixed problem size.

The situation is further highlighted by the weak scalability analysis in Figure 6, where execution time increases dramatically as more nodes are added to solve a proportionally larger problem.

4. Varying Payload Size

All previous analyses were conducted using files containing records with a maximum size of 64 bytes. Here we show how the distributed algorithm scales with a maximum size of 16 bytes instead, while maintaining the overall size of the file close to the original analysis.

Figure 7 shows marginally better strong-scaling results, especially moving from a 2-node to a 4-node configuration, but eventually flattens out indicating similar I/O saturation.

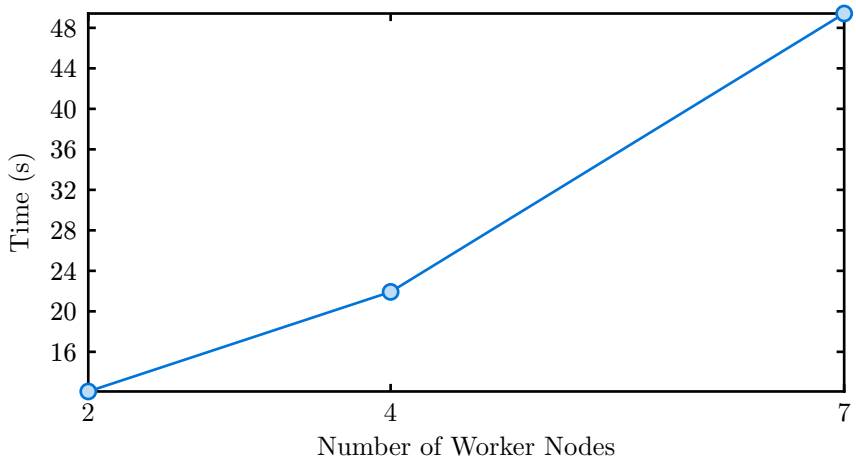

Figure 6: Weak scalability analysis of the distributed MPI implementation. The execution time is measured as both the number of worker nodes and the total problem size are increased proportionally, maintaining a constant workload per node. The ideal result would be a flat horizontal line, but the plot shows significant degradation, with communication and I/O costs dominating.

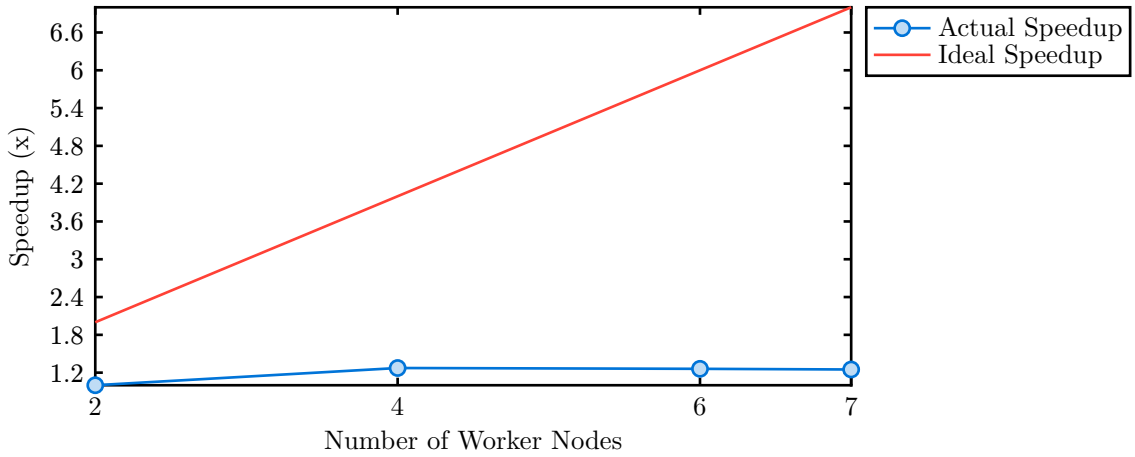

Figure 7: MPI strong scalability with a maximum payload of 16 bytes.

5. Cost Model Analysis

An approximate cost model for the distributed implementation of the outlined merge sort algorithm can be provided by employing the BSP model. Because the implementation introduces a global synchronization barrier between the two phases of the algorithm, we can independently focus our analysis on each of them.

We define the following quantities:

- N is the total number of records to sor
- B is the batch size that each worker node receives
- R is the average payload length in bytes
- T_{read} is the time required to read a single record from the NFS
- T_{write} is the time required to write a single record to the NFS
- p is the number of worker nodes
- g is the cost of sending a single word through the network

Both T_{read} and T_{write} can be implicitly scaled by the operating systems page size and the size of the buffers used to implement I/O operations.

5.1. Superstep 1

During phase 1, the first superstep consists of the emitter node performing a scan of the input file and scattering batches of records to the worker nodes:

$$T^{(1)}_{\text{Emitter}} = NT_{\text{read}}$$

Since sorting and writing to disk can be done employing stream parallelism, each worker nodes incur a cost of:

$$T^{(1)}_{\text{Worker}} = \frac{N}{Bp}\max(B\log(B), BT_{\text{write}}) = \frac{N}{p}\max(\log(B), T_{\text{write}})$$

and the first superstep's computation cost is thus $T^{(1)}_{\text{comp}} = T^{(1)}_{\text{Emitter}} + T^{(1)}_{\text{Worker}}$ because even though I/O latencies can in theory be hidden behind the worker's computations, NFS contention eventually serializes nodes operations.

The cost of the first superstep's communication is $h = NRg + l$.

5.2. Superstep 2

After the input file has been consumed, the worker nodes can merge their local sorted runs into a single output file. This is done in parallel, but again NFS contention is the primary bottleneck:

$$T^{(2)} = T^{(2)}_{\text{Worker}} = \frac{N}{p} T_{\text{read}} + \frac{N}{p} \log\left(\frac{N}{Bp}\right) + \frac{N}{p} T_{\text{write}} + l$$

No additional communication is required in this step

5.3. Superstep 3

The final superstep involves the emitter node performing a final merge pass over p sorted runs stored on disk, without additional communication requirements:

$$T^{(3)} = T^{(3)}_{\text{Emitter}} = NT_{\text{read}} + N\log(p) + NT_{\text{write}} + l$$

5.4. Total cost

The total cost is therefore

$$\begin{aligned} T_{\text{total}} &= T^{(1)} + T^{(2)} + T^{(3)} \\ &= NT_{\text{read}} + \frac{N}{p}\max(\log(B), T_{\text{write}}) + NRg + l \\ &+ \frac{N}{p} T_{\text{read}} + \frac{N}{p}\log\left(\frac{N}{Bp}\right) + \frac{N}{p} T_{\text{write}} + l \\ &+ NT_{\text{read}} + N\log(p) + NT_{\text{write}} + l \end{aligned}$$

Rearranging we get

$$T_{\text{total}} = \underbrace{T_{\text{read}}\left(2N + \frac{N}{p}\right) + T_{\text{write}}\left(N + \frac{N}{p}\right)}_{\text{I/O}} + \underbrace{\frac{N}{p}\max(\log(B), T_{\text{write}})}_{\text{Possibly hidden I/O}} + \underbrace{\frac{N}{p}\log\left(\frac{N}{Bp}\right) + N\log p}_{\text{Computation}} + \underbrace{NRg + 3l}_{\text{Communication}}$$

5.5. Analysis

As can be seen, this approximate cost model matches the empirical evidence collected by running the experiments on the cluster machine: the largest term is the I/O cost, with multiple full read and write passes over the entire dataset. Notice how this term barely changes as the number of nodes p increases. Depending on the batch size and the NFS contention, the second term can possibly hide the I/O cost of writing sorted runs on disk. The computation cost correctly scales with the number of nodes, but notice how the final merge phase cost ($N \log p$) actually increases with the number of processors used, consolidating our weak-scaling findings. Finally, the communication cost scales linearly with the size of the input dataset.

6. Conclusions

The experiments carried out on the *spmcluster* provided concrete evidence of the heavy bottlenecked I/O that the distributed algorithm goes through. All tested configurations performed poorly on a medium size dataset that could in theory be stored in internal memory and did not show any scaling capacity. The provided implementations for a shared memory system outperformed their distributed counterpart by a large margin, but still showed a modest and almost flat speedup over the number of threads used, with decreasing efficiency as the number of threads went up. The empirical and theoretical analysis showed that the main bottleneck is the centralized shared NFS, which created extremely high contention over the nodes, dwarfing any introduced parallelism.

Implementation	N=1	N=2	N=4	N=6	N=8
OMP	8169	7065	6686	6824	6890
FF	9471	9410	7907	7551	7060

Table 1: Raw execution times (in milliseconds) for the two shared-memory implementations running on a single *spmcluster*'s node with a varying amount of available threads.

Implementation	N=2	N=4	N=6	N=7
MPI + OMP	12086	11661	10994	11581

Table 2: Raw execution times (in milliseconds) for the distributed implementations running on a varying amount of nodes.

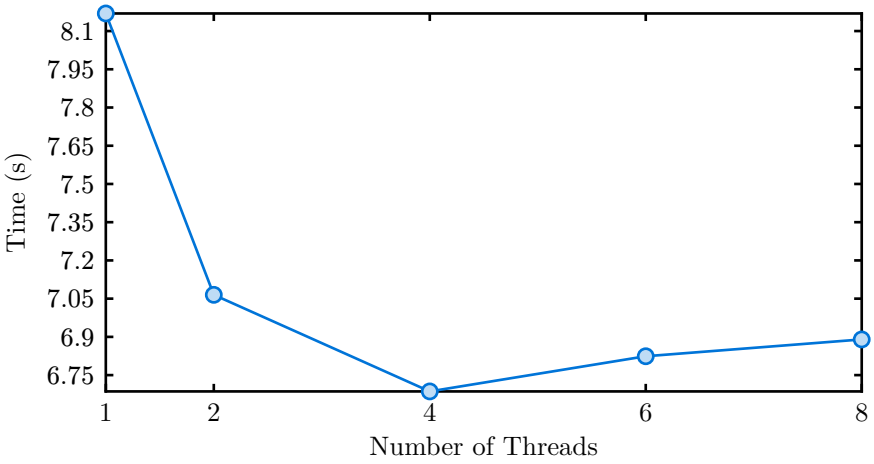

Figure 8: Execution time of the OpenMP implementation on a single node. Performance improves up to 4 threads, after which the overhead of thread management and I/O saturation begins to dominate, leading to diminishing returns.

A. Raw execution times

This appendix reports raw execution time plots and tables for all the provided implementations.

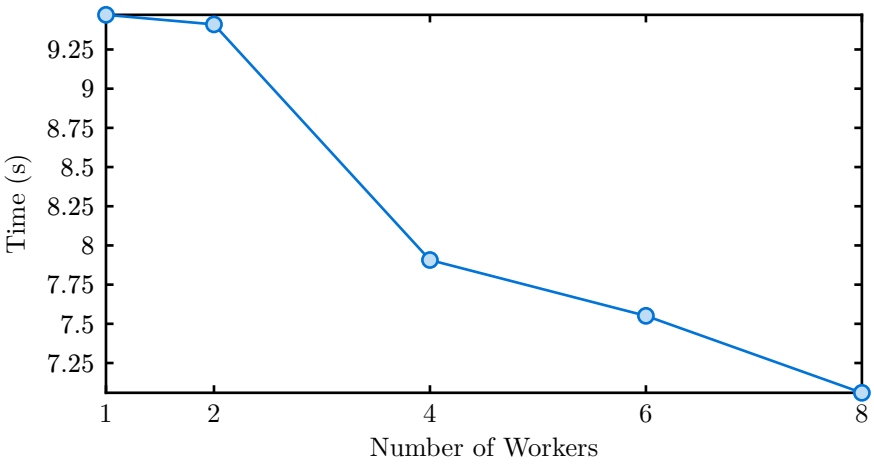

Figure 9: Execution time of the FastFlow implementation. A notable performance improvement is visible when scaling from 2 to 4 workers, as the pipeline parallelism effectively hides I/O latency. Gains diminish with more workers due to the same underlying I/O bottleneck.

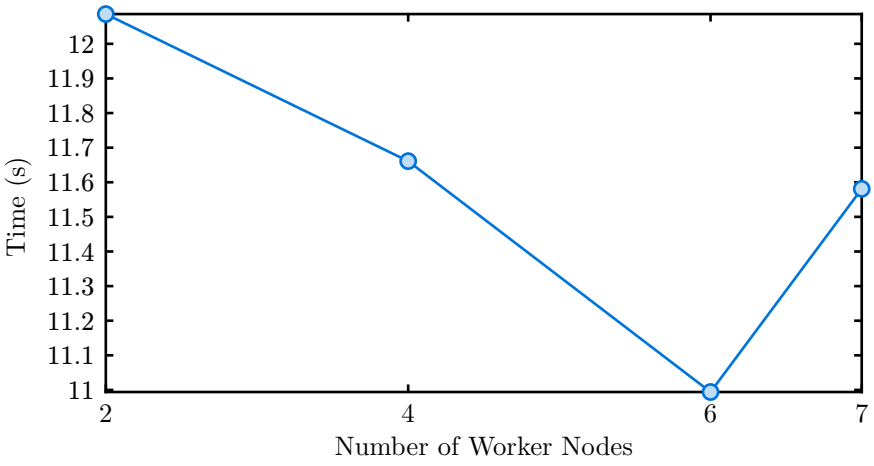

Figure 10: Execution time of the distributed MPI implementation, varying the number of worker nodes. Each node utilized 8 OpenMP threads. The performance gains are minimal, and performance degrades beyond 6 nodes, indicating high communication overhead.