SPM Project Report

Leonardo Scoppitto
№ 545615

Contents

Introduction

The goal of this project is to implement and assess the performance of the parallelized external mergesort algorithm using **OpenMP** and **FastFlow** for the single node version and **MPI + OpenMP/ FastFlow** for the hybrid distributed version.

Project structure

Here's the project structure:

- `src`: contain all the source code, the `fastflow` library source and the `include` dir with the `.hpp` files.
- `scripts`: a python script to plot the results of a run for a quick comparison and to compute the speedup.
- `results`: the log files containing the execution times.
- `benchmark.sh`: the script to execute a benchmark. It can be configured to run both locally and on the cluster using `srun`. The files `cluster_run.sh` and `local_run.sh` are two helper scripts with the proper configuration of the benchmark script based on where it has to run.
- `Makefile` containing the compilation commands.

Execute the project

`local_run.sh` and `benchmark.sh` take 3 parameters as input:

1. `input_file`: the path where to generate the test file. All the intermediate files will be generated in the same directory
2. `max_payload_size`: the maximum size, in bytes, that the records payload can have.
3. `number_of_records`: the number of records to generate.

Example execution on your local machine:

```
./local_run.sh /tmp/test.dat 16 10000000
```

To test the project quicker on the cluster, the execution parameters have been hard coded in the `cluster_run.sh` file:

```
./cluster_run.sh
```

To manually compile the project and test one of the implementations:

```
# If needed, load the openmpi module
module load mpi/openmpi-x86_64

# Compile
make clean
make -j $(nproc)

# Generate test file and execute
./gen_file -r payload_size -s n_records /path/to/file
./mergesort_ff -t nthreads -m max_usable_mem_bytes /path/to/file
# All the other executable share the same parameters
```

To parse and read a log file:

```
# These two libraries are required
pip install plotly numpy

# Point to the log file without the _mpi* suffix
PLOT=1 python3 scripts/plot.py /path/to/logfile
```

1 General implementation

The external mergesort algorithm execution is split in two parts:

- The **creation** of the sorted sequences
- The **merge** of the sorted sequences

Generating the sequences

To tackle this problem, two approaches have been explored:

1. The first naive approach is to read as many bytes as can fit into memory and create a vector of `Record`, sort the vector in place using `std::sort` and then flush the sorted records to disk.
2. Use the *snow plow*[1] technique to try to the maximize the length of the sorted sequences.

The differences between the two approaches have been tested on both fast and slow storage, and in both cases the naive approach proved to be better in almost every situation: the snow plow technique requires both a lot of insertions and extractions from an heap and a lot of small reads and writes, which, even if buffered, overwhelm the gains during the merge phase. For this reason the `std::sort` approach has been adopted[2].

Merging the sequences

For the merging phase both binary merge and *k-way merge* algorithms have been evaluated and, while the merging algorithm can be selected in the sequential version using the flag `-k`[3] , in the parallel version the k-way merge is enabled by default as it is way more performant than the binary merge. In the parallel versions, when the last sequence has been generated, each thread picks their chunk of files to merge together and produces one file. At this point, the remaining files (equal to the number of threads) will be merged by the main thread using, again, k-way merge.

Memory usage

To be able to test the external sorting algorithm without requiring files larger than the machine's memory, the parameter `-m MAX_MEM` was introduced to limit the memory that the application can use. This functionality has been implemented by making each thread responsible for tracking the bytes read and written from and to disk, paying attention not to read more than assigned memory quota. In the sequential version, of course, the main thread can use up to `MAX_MEM` bytes, while in the parallel version each thread has assigned `MAX_MEM / nthreads` memory.

[1] See appendix A.1.

[2] The snow plow code can still be consulted under `src/include/sorting.hpp::genSequenceFiles`.

[3] The `-k` option enables the k-way merge, but for the sake of simplicity the number of sequences to merge together is fixed and is set to the number of the number of the sequences.

2 Parallel implementation

2.1 Single node

Both implementations of the parallel algorithm follow the same structure:

1. A single thread reads the file to be sorted in chunks[4] to compute the first and the last byte positions of a chunk `chunk_size` long. Then, those boundaries are passed to a worker thread that will re-read the chunk (maximum `MAX_MEM / nthreads` bytes at a time) to generate the sorted sequence files. This proved to be more efficient than:
   - Reading about `MAX_MEM / nthreads` bytes and sending them to a worker thread.
   - Perform a lot of small reads to parse the record length and skip the key and the payload with `lseek`.
2. At this point, the sorted runs will be merged in parallel by each thread using the k-way merge algorithm.
3. When all the k-way merges are done, the main thread will execute the k-way merge on the sequences produced in the step 2.
4. At the end, the last merged sequence is renamed to `output.dat`

OpenMP

The implementation of the OMP version can be consulted in `src/include/omp_sort.hpp`.

The sorting phase of the OpenMP version[5] has been implemented using an `omp parallel` region to spawn the threads along with an `omp single` region:

1. One thread accesses the disk to compute the chunks bounds.
2. The task represented by the byte offset, the chunk size and the uuid of the file to generate is scheduled using the directive `omp task firstprivate(start_offset, size, uuid)`, in order to pass a copy of the parameters computed in step 1 to the worker thread.
3. Once all the chunks have been submitted, the main thread waits for the tasks to complete using `omp taskwait` and then the list of the generated sequences is returned.

At this point the vector containing the sorted files path to merge is passed to `src/inlcude/omp_sort.hpp::ompMerge`, which splits the vector into chunks of files that will be merged in parallel using the directive `parallel for`.

Finally, the main thread merges the resulting intermediate sequences, completing the execution of the algorithm.

FastFlow

The implementation of the FF version can be consulted in `src/include/ff_sort.hpp`.

The algorithm has been implemented using a farm (`ff_farm`) with a `Master` node that acts as both the emitter and the collector and multiple `Worker` nodes:

1. The `Master` quickly scans the file to divide it into chunks like in the OMP implementation, sending out a `sort_task` to a `Worker` node when the chunk is large enough[6].

[4] The memory constraint is respected using a circular buffer.

[5] See `src/include/omp_sort.hpp::genRuns`.

[6] See `src/include/ff_sort.hpp::Master::send_out_sort_tasks`

2. The `Worker` node that receives the task starts generating the sorted files that will be later merged. When the generation has finished, the `Worker` returns its original task definition to the `Master`, which now contains the list of the files produced.
3. The `Master` stores all the files produced by the workers in a vector (where the i-th position corresponds to the i-th worker). When a `Worker` finishes all the submitted tasks, the `Master` sends out a `merge_task` to that `Worker`, so it can start the merging phase on his files.
5. When the `Master` gets back all the merge tasks, it merges the remaining chunks using the k-way merge, like in the OMP version.

This implementation has been tested with thread pinning and blocking mode both on and off[7] and, in most situations the algorithm performed better with thread pinning off and blocking mode on, being the workload mostly IO-bound. A detailed comparison can be consulted in section 4.1.1.

2.2 Multiple nodes (MPI + OpenMP)

The distributed version of the algorithm followed a slightly different approach:

1. The master node (rank 0) reads the file in chunks that will be sent to the worker nodes (rank ≥ 1) as an array of `MPI_CHAR`. The master sends the size first, then the data.
2. The worker receives the size and then the payload that will be de-serialized in a `vector<Record>`. When there are enough elements in the vector, it is sorted using `std::sort`[8] and then flushed into a temporary file on the worker file-system[9].
3. When the master reaches the `EOF`, it sends a size equal to 0 to all the worker to signal that they can start merging the sorted files.
4. The workers merge the file using `src/include/omp_sort.hpp::ompMerge` and then they start sending the resulting file to the master.
5. Using OMP, the master spawns a thread for each worker and collects the sorted runs in parallel.
6. The master merges the runs using the k-way merge algorithm.

OpenMP was chosen because the existing `ompMerge` method could be reused directly without requiring modifications. In contrast, to use the the FastFlow implementation as-is would have forced each worker node to first write its assigned chunk to a file and then execute the algorithm on that file, introducing additional IO overhead.

[7] The thread pinning is disabled by default calling `farm.no_mapping()` before running the farm. It can be re-enabled using the parameter `-x`. The blocking mode can be enabled using the flag `-y`.

[8] First, I tried to use an heap to keep a sorted collection of records, but the performance were slightly worse than `std::sort`.

[9] It defaults to `/tmp` being the home folder of the cluster nodes an NFS mount, but it is settable with the `-p /path` parameter.

3 Cost model

Defined:

- N: number of records
- r: the max payload size for each record
- $B = N \cdot r$: input size in bytes ← upper bound
- b: chunk size sent to workers (bytes)
- $C = \frac{B}{b}$: number of chunks
- t: number of OMP threads per worker
- $p \geq 2$: total MPI processes (nodes)
- g: communication cost
- l: latency
- $T_{\text{read}}, T_{\text{write}}$: costs to perform the $_{\text{operation}}$

The algorithm implemented using MPI can be modeled using the BSP model, identifying 3 supersteps:

1. The master reads the input file in chunks and sends them to the workers.
2. The workers merge the sorted files locally and send back the merged result to the master.
3. The master merges the final sequences to produce the output file.

Superstep 1

- **Master:**
  - The cost of reading the whole file is $C \cdot b \cdot T_{\text{read}} = B \cdot T_{\text{read}}$
  - The communication cost to send the chunks to the workers is $T_{\text{comm}} = C \cdot (l + b \cdot g)$
- **Workers:**
  - The cost to sort the chunks is: $T_{\text{sort}} = \frac{C}{p-1} \cdot \frac{b}{r} \cdot \log\left(\frac{b}{r}\right)$
  - The cost of writing the sequences to the temporary location is : $\frac{C}{p-1} \cdot b \cdot T_{\text{write}}$

So the cost of this superstep is:

$$T_{\text{S1}} = B \cdot T_{\text{read}} + C \cdot (l + bg) + \frac{C}{p-1} \cdot \left(\frac{b}{r} \cdot \log\left(\frac{b}{r}\right) + b \cdot T_{\text{write}}\right)$$

When there is more than one worker node ($p > 2$), it is possible that some of the IO cost on the master and the communication cost overlap to the computation on the workers. Considering that, the cost can be (optimistically) reduced to:

$$T_{\text{S1}} = \underbrace{b \cdot (p-1) \cdot T_{\text{read}}}_{\text{Initial read cost}} + \underbrace{(p-1) \cdot (l + bg)}_{\text{Initial comm cost}} + \underbrace{\frac{C}{p-1} \cdot \left(\frac{b}{r} \cdot \log\left(\frac{b}{r}\right) + b \cdot T_{\text{write}}\right)}_{\text{Sorting + writing sequences to disk}}$$

Here I considered that once every worker has been fed a chunk, the master can continue to read the input files and submit them to the next nodes with a round robin policy while the workers are sorting the files.

Superstep 2

Each worker merges at most $\frac{C}{p-1}$ sequences of size b, so it process $\frac{B}{p-1}$ bytes:

- The IO cost is $\frac{B}{p-1} \cdot (T_{\text{read}} + T_{\text{write}})$ as it happens in parallel on each worker's filesystem.
- Defined $n = \frac{B}{r \cdot (p-1)}$ the number of records that every worker has to process, the computational cost for the parallelized k-way merge[10] is:

$$T_{\text{merge}} = \begin{cases} \frac{n}{t} \cdot \log\left(\frac{C}{t \cdot (p-1)}\right) + n \cdot \log(t) & \text{iff the number of sequences is } \geq 2t \\ n \cdot \log\left(\frac{C}{p-1}\right) & \text{else} \end{cases}$$

- Since the master can only accept up to MAX_MEM bytes to respect memory constraints and since it receives from all the workers concurrently, each worker can send up to $\frac{\text{MAX\_MEM}}{p-1}$ bytes per round. Defined $M = \text{MAX\_MEM}$, the communication cost to send the merged sequence back is then $T_{\text{comm}} = \max\left\{\frac{B}{M}, 1\right\} \cdot (p-1) \cdot l + \frac{B \cdot g}{p-1}$ since the master receives all the sequences in parallel.
- Finally, the master has to write all the sequences to the disk: $B \cdot T_{\text{write}}$.

Since there is almost no overlap of computation and communication, the cost of the second superstep is:

$$T_{\text{S2}} = \frac{B}{p-1} \cdot (T_{\text{read}} + T_{\text{write}}) + T_{\text{merge}} + \underbrace{T_{\text{comm}} + B \cdot T_{\text{write}}}_{\substack{\text{These two terms could} \\ \text{overlap a bit}}}$$

Superstep 3

Again, the master has to read and write all the sequences, which costs $B \cdot (T_{\text{read}} + T_{\text{write}})$, while the merge cost is $\frac{B}{r} \cdot \log(p-1)$. So the last superstep cost is equal to the sequential k-merge[11] plus the cost of IO:

$$T_{\text{S3}} = B \cdot \left(T_{\text{read}} + T_{\text{write}} + \tfrac{1}{r} \cdot \log(p-1)\right)$$

Total cost

$$\begin{aligned} T_{\text{tot}} = b \times (p-1) \cdot T_{\text{read}} + (p-1) \cdot (l + bg) + \tfrac{C}{p-1} \cdot \left(\tfrac{b}{r} \cdot \log\left(\tfrac{b}{r}\right) + b \cdot T_{\text{write}}\right) + \\ + \tfrac{B}{p-1} \cdot (T_{\text{read}} + T_{\text{write}}) + T_{\text{merge}} + T_{\text{comm}} + B \cdot T_{\text{write}} + \\ + B \cdot \left(T_{\text{read}} + T_{\text{write}} + \tfrac{1}{r} \cdot \log(p-1)\right) \end{aligned}$$

Let's analyze each superstep:

1. Considering the "optimistic" approximation made, the first two terms in the first superstep are directly related to the number of workers, increasing the cost if nodes are added, as more communication is needed before the workers computational cost overlaps with the master IO and comm costs. The last term of this superstep, on the other hand, is inversely proportional to the number of workers and, independently from the considerations above, it reduces as more workers are added.
2. The first term of the second superstep, just like in the previous one, is inversely proportional to the number of workers, since the more workers you have, the less data they have to process and the IO happens on the worker's filesystem, so it is parallelized. Regarding the merge cost, the term $(p-1)$ is at the denominator of both the terms outside and inside the logarithm, correctly bringing down the cost of the merge as more nodes are added. However, the two terms in the communication cost behave differently one from the other: the latency increases with the number of worker since more messages are needed, while the message payload decreases.

[10] The sequential algorithm has been implemented using an heap so the complexity is $T(n,k) = 2n \times \mathcal{O}(\log k) = \mathcal{O}(n \log k)$.

[11] To be precise, the `ompMerge` method is used, but the condition $2 \cdot t \geq p$ (where t is the number of threads and p the number of nodes) is never `true`, at least in the tests conducted. In any case, it's not worth merging few files in parallel.

3. Here, the algorithm is executed only by the master, so the only term related to the number of workers is the number of sequences to merge, that impact the cost of the k-way merge algorithm: more nodes, more sequences to merge, thus the cost increases.

4 Test results and performance analysis

The tests have been conducted on both the spmcluster and my private server[12], especially the more specific ones, in order not to overcrowd the cluster.

The speedup has been computed using the best sequential algorithm results (usually the k-way mergesort).

Note: In every execution the maximum memory usable is always set to $\frac{1}{10}$ of the file size.

4.1 Single node results

4.1.1 Results premise - FastFlow fine tuning

After the first local tests, the performance of FastFlow, especially on faster storage, left me disappointed in terms of peak speedup. So, I tried to disable the thread pinning using the `no_mapping` method and I obtained a substantial performance gain in almost all situations.

Then comparing it to OpenMP I noticed that when the thread count is higher than the physical core count (the processors on the server have hyperthreading enabled), the speedup dropped abruptly.

To try to mitigate this behavior, I enabled the FastFlow blocking mode[13], which disables busy waiting on the message channels: being the workload IO-bound, disabling busy waiting could be useful as often both the `Master` node and the `Worker` nodes have to wait each other while performing IO operations. Also, for higher thread count (i.e. when the thread count is higher than the physical core count) it can be useful as it reduces the contention for the same physical core resources. This improvement can be seen in the graphs below.

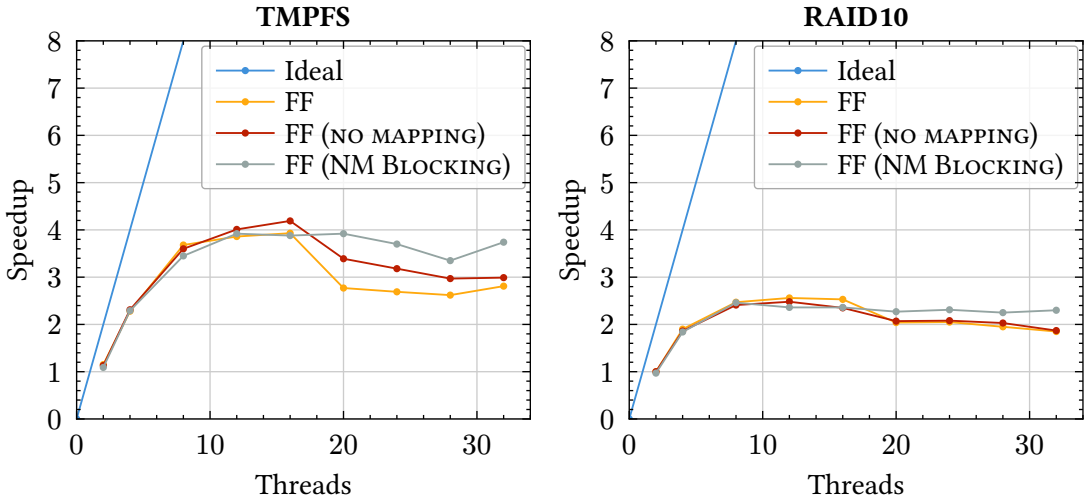

Figure 1: Comparison of the performance gains on tmpfs (*left*) and on RAID10 (*right*). These preliminary performance assessments have been conducted on the **private server**.

In any case, in the following sections, all the FastFlow executions are plotted to highlight the performance differences with this kind of workload.

[12] Dell Poweredge R720 with two Intel E5-2650 v2 (8C/12T each) and 64GB RAM.

[13] Using the `ff_farm::blocking_mode()` method.

4.1.2 Cluster results

In the TMPFS run, where the IO overhead is low, both implementations achieved approximately the same peak speedup when the thread count matched the number of physical cores. Beyond this point, FastFlow performance began to degrade if blocking mode was disabled, whereas OpenMP continued to scale, but at a lower rate. Nevertheless, the efficiency declined rapidly in both cases, reflecting the IO-bound nature of the workload.

On NFS, FastFlow with blocking mode showed more consistent results, though this may be partly due to lower network traffic at the time those tests were executed.

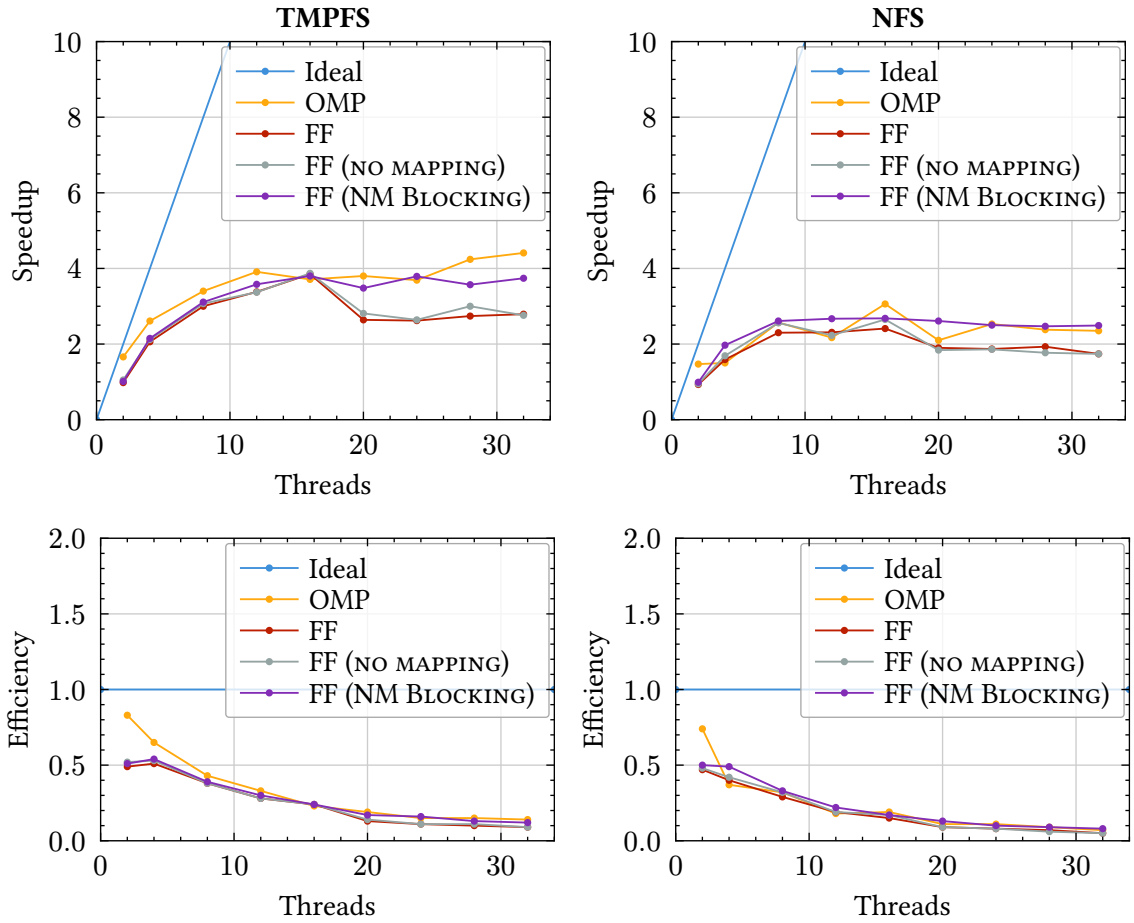

Figure 2: Payload **size** is **64 bytes**, records **count** is **50M**.

Threads	OMP	FF (NM)	FF (BL)
2	1.66	1.05	1.01
4	2.61	2.12	2.15
8	3.4	3.07	3.11
12	3.91	3.37	3.58
16	3.71	**3.87**	**3.8**
20	3.8	2.81	3.48
24	3.69	2.64	3.79
28	4.24	3	3.57
32	**4.41**	2.76	3.74

Threads	OMP	FF (NM)	FF (BL)
2	1.47	0.95	0.99
4	1.5	1.69	1.97
8	2.57	2.56	2.61
12	2.17	2.23	2.67
16	**3.06**	**2.65**	**2.68**
20	2.1	1.84	2.61
24	2.53	1.86	2.5
28	2.38	1.77	2.47
32	2.35	1.74	2.49

Table 1: Speedup table of the results presented above. TMPFS on the left, NFS on the right.

4.1.3 Varying payload size

To test how the algorithm behaved with different payload sizes, I run a benchmark[14] keeping the file size almost the same and varying the size of the payload and the number of records to sort, actually reducing the computational work and increasing the IO.

As we can see from the graphs below, the maximum speedup, and consequently the efficiency, decreased as the size grew. Also, note that the greater is the IO cost (in terms of payload size or IO bandwidth), the closer are the speedups between the OMP and FF implementations.

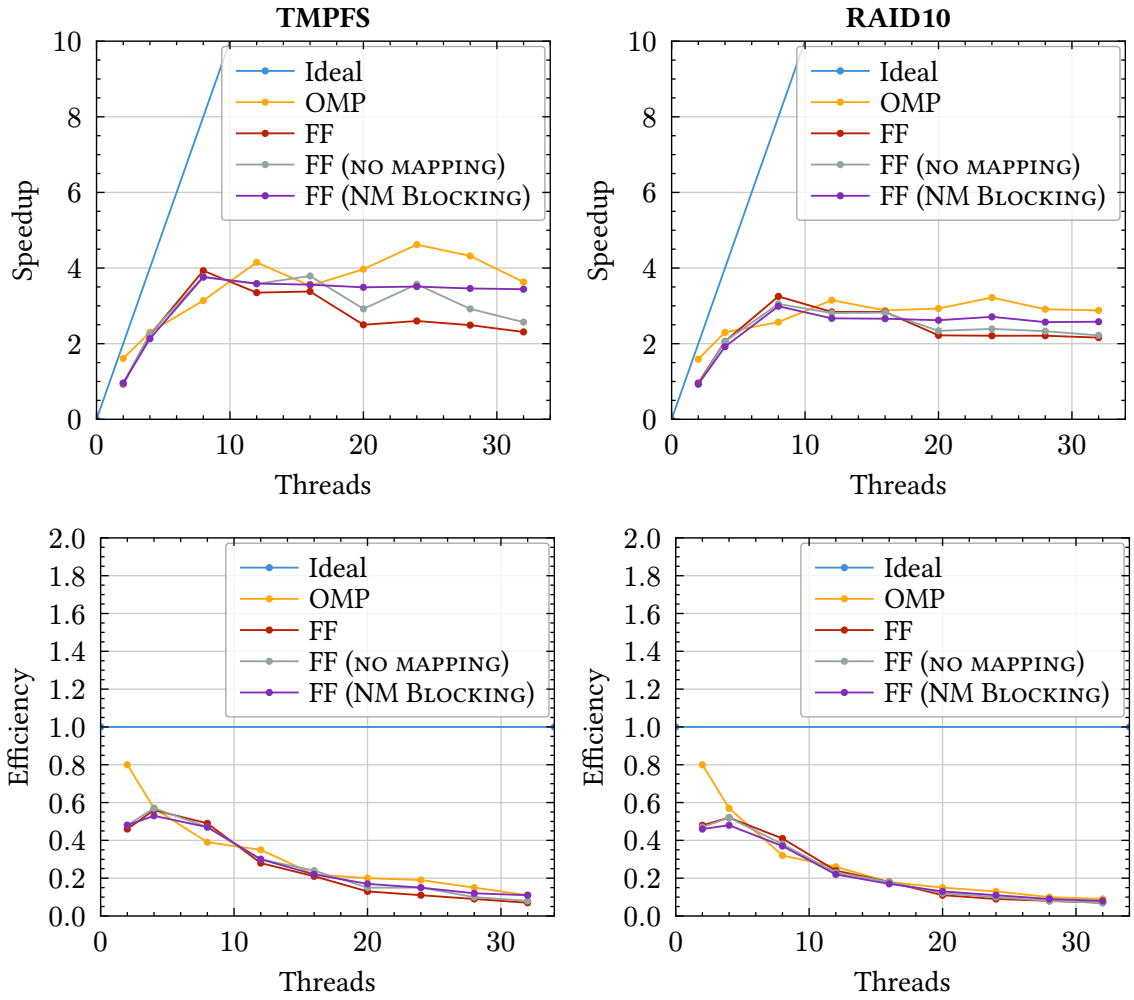

Figure 3: Payload **size** is **16 bytes**, records **count** is **20M**.

[14]The results provided in this section have been collected on the private server.

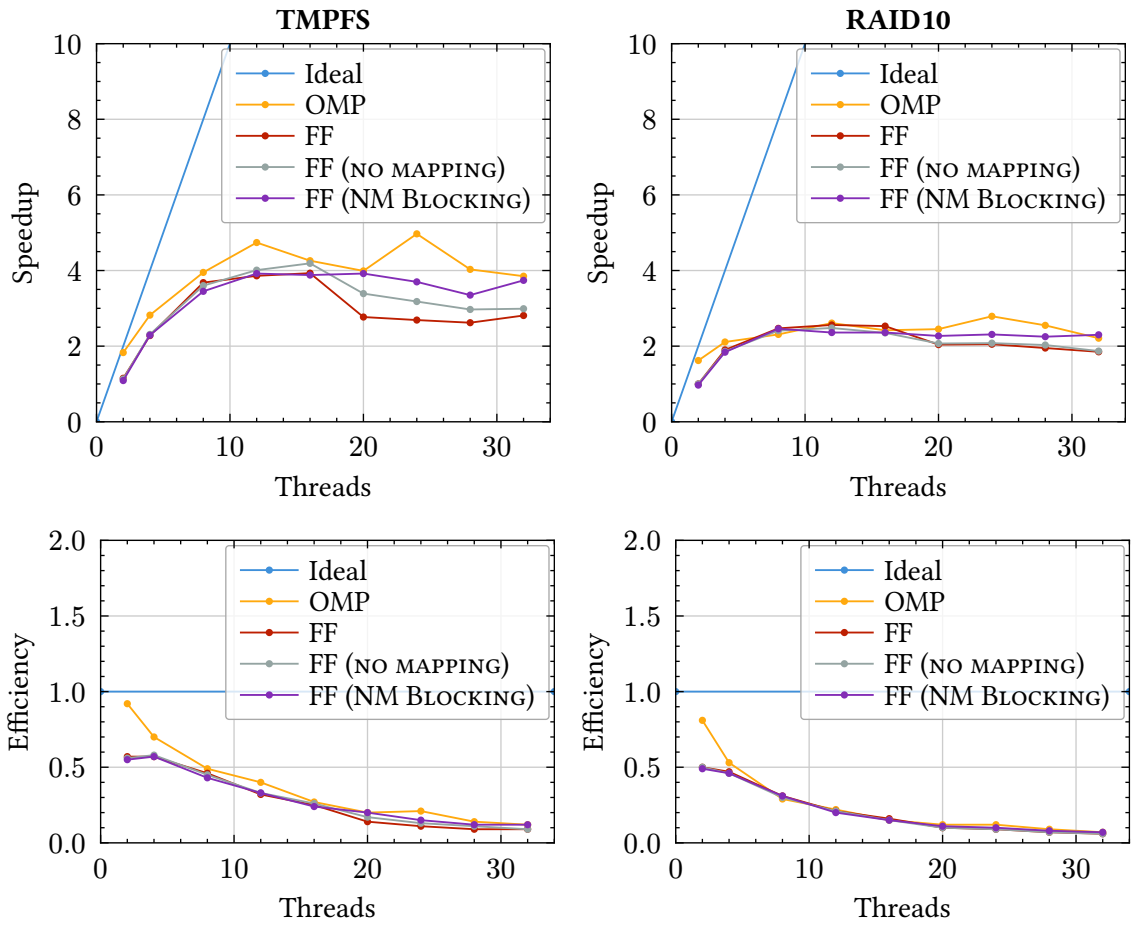

Figure 4: Payload **size** is **64 bytes**, records **count** is **10M**.

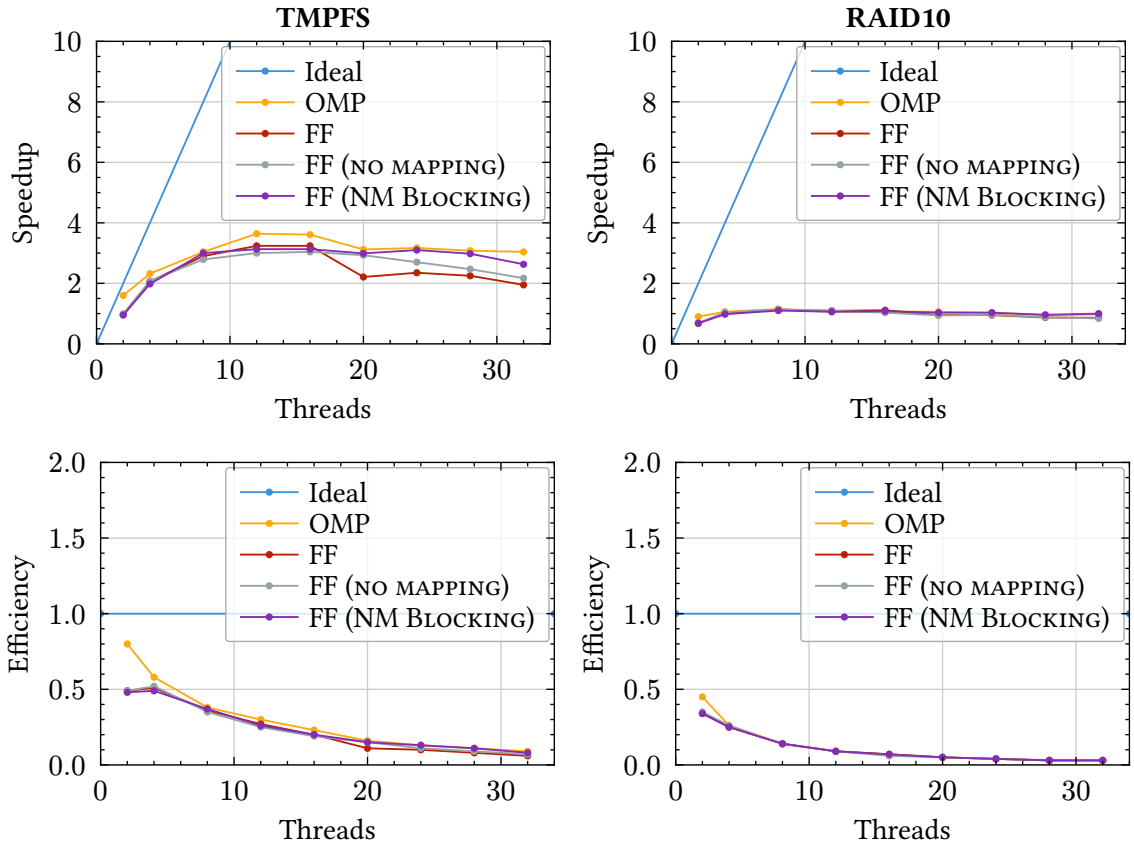

Figure 5: Payload **size** is **512 bytes**, records **count** is **12.5M**.

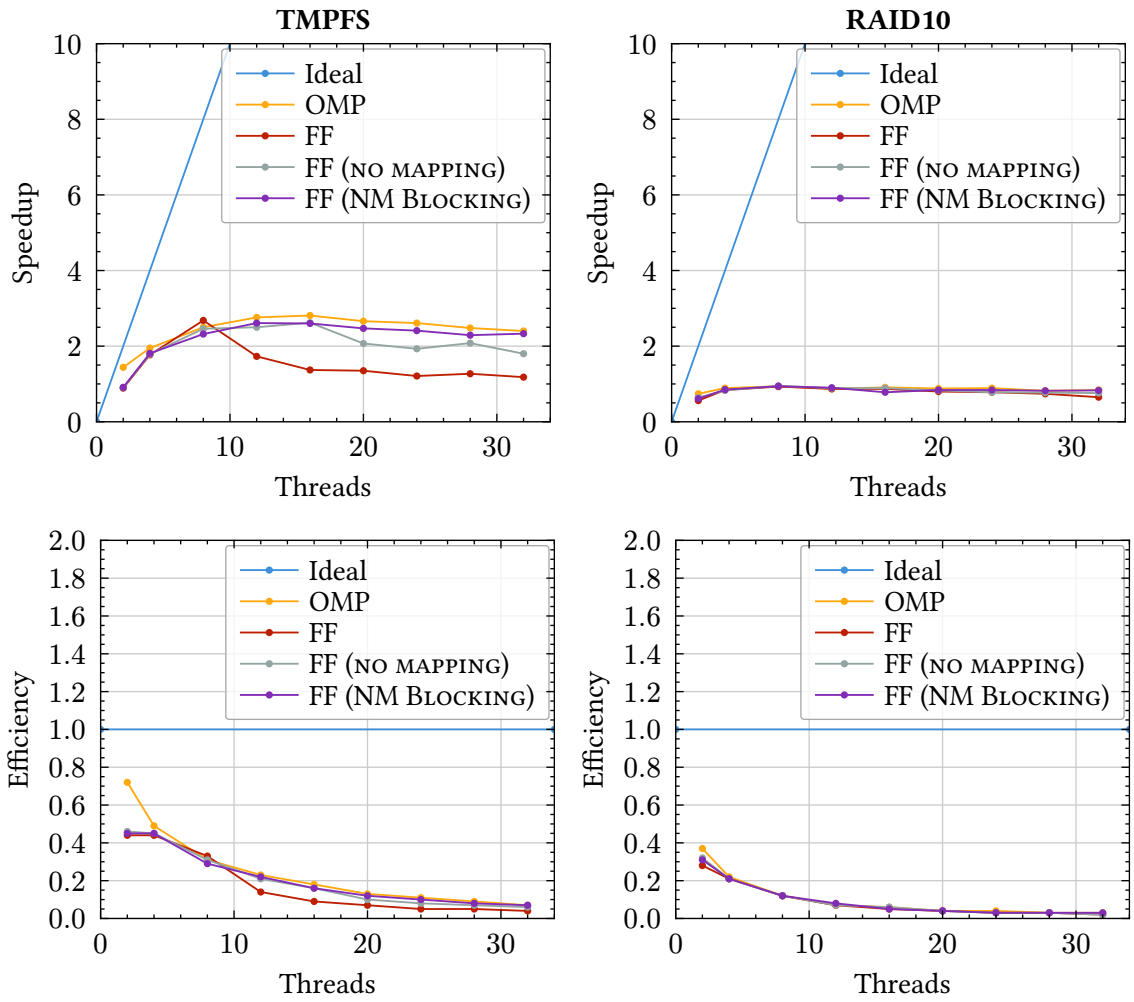

Figure 6: Payload **size** is **2048 bytes**, records **count** is **31250**.

4.1.4 OpenMP vs FastFlow comparison

Despite the similar implementation of the algorithm, there is a performance difference between the two. At first, I attributed this to possible automatic optimizations performed by OpenMP, but this alone did not fully account for the observed difference. Then I realized that the merge phase (which happens to be where the most work is done) has a key difference in the two implementations:

- In OpenMP a `parallel for` directive is used to merge the runs concurrently, so each thread get a chunk of the files to merge and runs the k-way merge on them.
- In FastFlow the `Master` node generates the merge tasks and then distributes them to the `Worker` nodes, meaning that he does not contribute actively to the merging phase.

To validate this observation, the variable `REDUCE_GAP` was introduced into the benchmarking script, allowing the `mergesort_omp` test to be executed with one fewer thread than the corresponding `mergesort_ff` run.

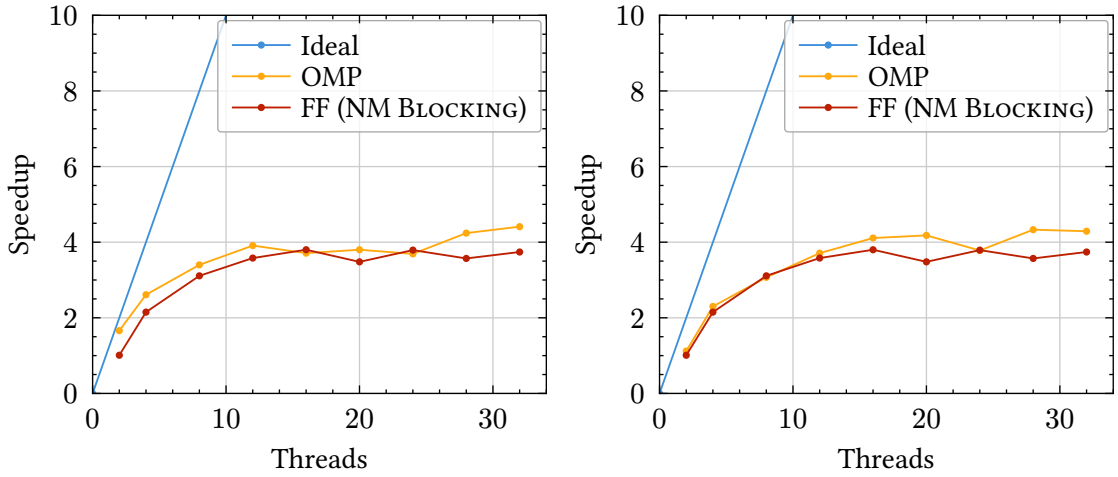

Figure 7: Comparison between the normal (*left*) and adjusted (*right*) runs on **cluster** tmpfs.

Normal run

Threads	OMP	FF (BL)
2	1.66	1.01
4	2.61	2.15
8	3.4	3.11
12	3.91	3.58
16	3.71	**3.8**
20	3.8	3.48
24	3.69	3.79
28	4.24	3.57
32	**4.41**	3.74

REDUCE_GAP set

Threads	OMP	FF (BL)
2	1.12	1.01
4	2.3	2.15
8	3.07	3.11
12	3.71	3.58
16	4.11	**3.8**
20	4.18	3.48
24	3.78	3.79
28	**4.33**	3.57
32	4.29	3.74

Table 2: Speedup table of results presented above.

As the plots indicate, when `REDUCE_GAP` is enabled, the speedup of OpenMP and FastFlow becomes comparable at lower thread counts.

4.2 Distributed algorithm results

The number of OMP threads for each run has been set to 32.

4.2.1 Strong scaling

As we can see from the plots, the performance gains in the distributed version are very poor since the most of the IO work is done by the master node and the execution requires a lot of data to be transmitted.

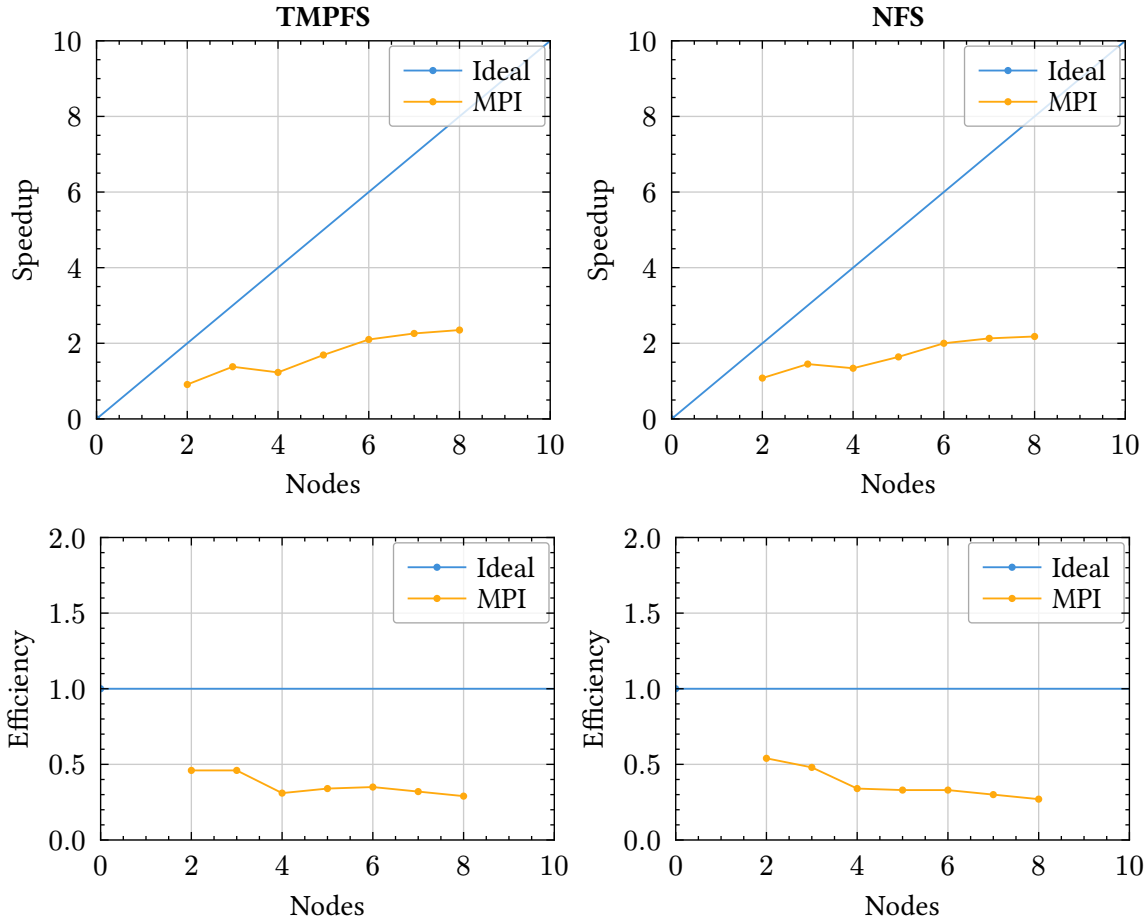

Figure 8: Payload **size** is **64 bytes**, records **count** is **50M**.

TMPFS

Nodes	MPI
2	0.91
3	1.38
4	1.23
5	1.69
6	2.1
7	2.26
8	**2.35**

NFS

Nodes	MPI
2	1.08
3	1.45
4	1.34
5	1.64
6	2
7	2.13
8	**2.18**

Table 3: Speedup table of the results presented above.

4.2.2 Weak scaling

To test the weak scalability, first, the application has been run on 2 nodes (one master and one worker, since it is the minimum required to execute it) with a max payload of 64 bytes and 50M items to gather the baseline data. Then the following MPI runs generated a file of $n \times BC$ where n is the number of nodes and BC is the base items count.

As shown in the graph below, although the number of nodes increases, the scale factor decreases. This indicates that the workload does not fully benefit from additional computing resources, as the primary bottlenecks are inter-node communication and the IO throughput of the master node.

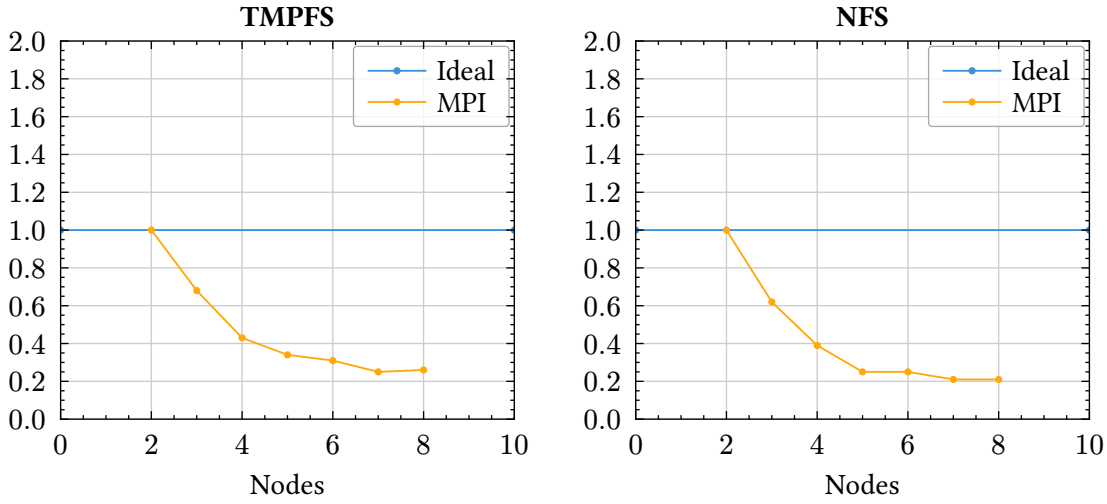

Figure 9: Weak scalability comparison between TMPFS and NFS.

Conclusions

The project has been a good opportunity not only to learn more about how to parallelize a workload using OpenMP, FastFlow and MPI, but also to experiment both with algorithms seen only in theory (e.g. the k-way merge or the snow plow technique) and with the optimizations that can be done to perform IO operations efficiently, such as using memory mapping and so on.

There are several aspects of the current implementations that could be further optimized:

- Implementing a custom allocator or a memory arena could reduce the overhead of frequent allocations and de-allocations during record serialization and de-serialization, since profiling results indicate that a significant portion of execution time is currently spent in `malloc` and `free`.
- If the memory constraint can be removed, some operations can be easily refactored to be more efficient, such as:
  - Introducing a collector node in the FastFlow implementation that collects the sorted `Records` vectors from the workers and writes them to disk, allowing some overlap between the computation of the sequences/merges and the IO operations.
  - Check whether with less memory constraints and with a more efficient memory management, the snow plow algorithm could become a viable option, thus reducing the cost of the merge phase.

omit

Appendix

A.1 SnowPlow

Here's the outline of the snow plow[15] technique:

Algorithm 1: A phase of the snow plow algorithm

```
1:  procedure SnowPlow(𝒰, InputSequence)
2:      ▷ Build ℋ as a min-heap over 𝒰's items
3:      𝒰 ← ∅
4:      r ← n
5:
6:      while ℋ ≠ ∅ do
7:          min ← top(ℋ)
8:          ▷ Write min to the output run
9:          next ← Read next from InputSequence
10:         if next < min then
11:             ▷ insert next in 𝒰
12:         else
13:             ▷ insert next in ℋ
14:         end
15:     end
16: end
```

[15] Source: *Pearls of Algorithm Engineering* - P. Ferragina.