h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences
– Faculty of Computer Science –

Preliminary lab report for the third high-performance computing exercise

by
Zebreus

INTRO

In the third lab we will try to run a distributed sorting algorithm using openMPI. We are using Rust and MPI to implement the third lab.

NON-DISTRIBUTED IMPLEMENTATION

We wrote a non-distributed implementation and compared it to the reference CPP implementation. We tried various sorting algorithms, but decided to focus on radix sort and the standard libraries `sort_unstable()`.

DISTRIBUTED IMPLEMENTATION

Our distributed implementation splits the input into 256 buckets based on the first byte of an entry. For each bucket there is exactly one responsible worker node. A worker node can be responsible for sorting multiple buckets.

The manager node reads the whole input, splits it into buckets, and streams the buckets to the worker nodes. After all buckets have been sent, the manager node retrieves the sorted buckets from the workers and merges them into a sorted list. The manager node then writes the sorted list into a file. The exact operation of the manager node is shown in Figure 1. The client flow can be seen in Figure 2.

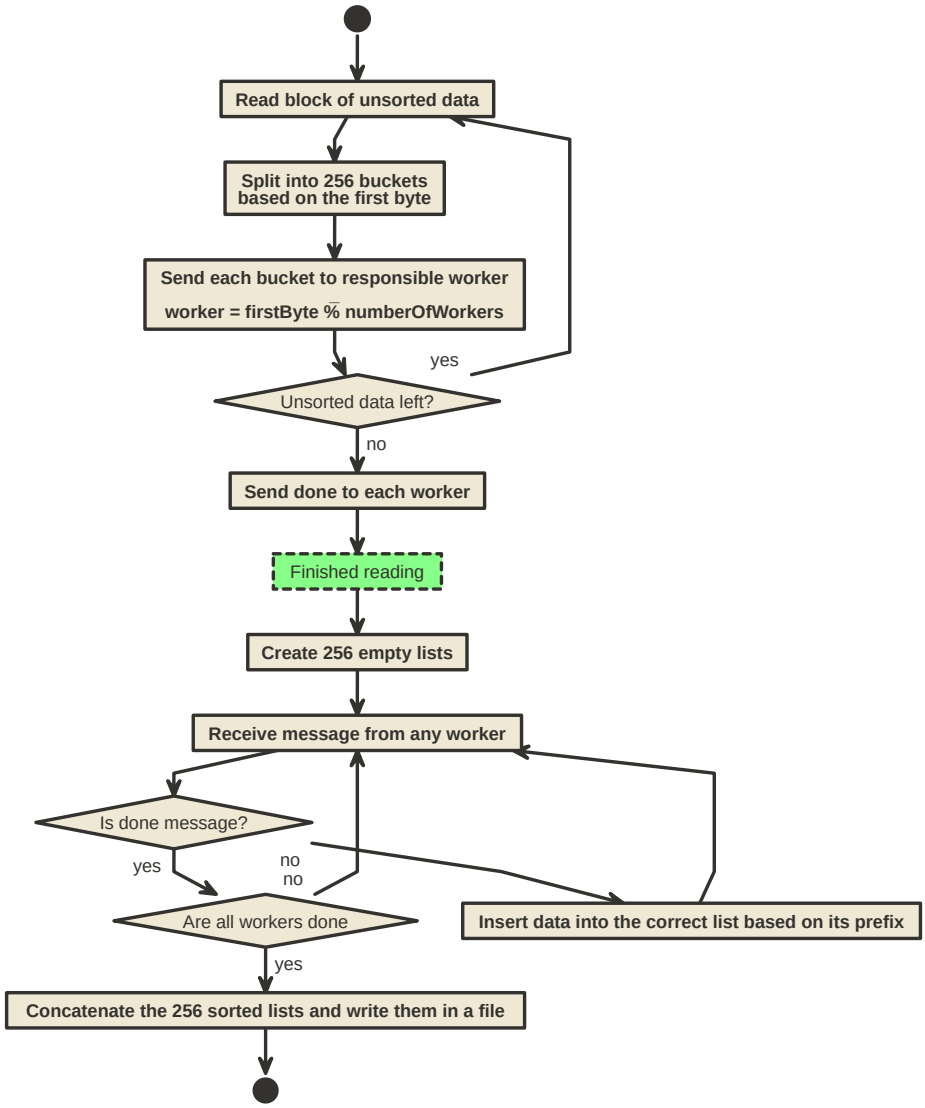

Figure 1. Manager flow

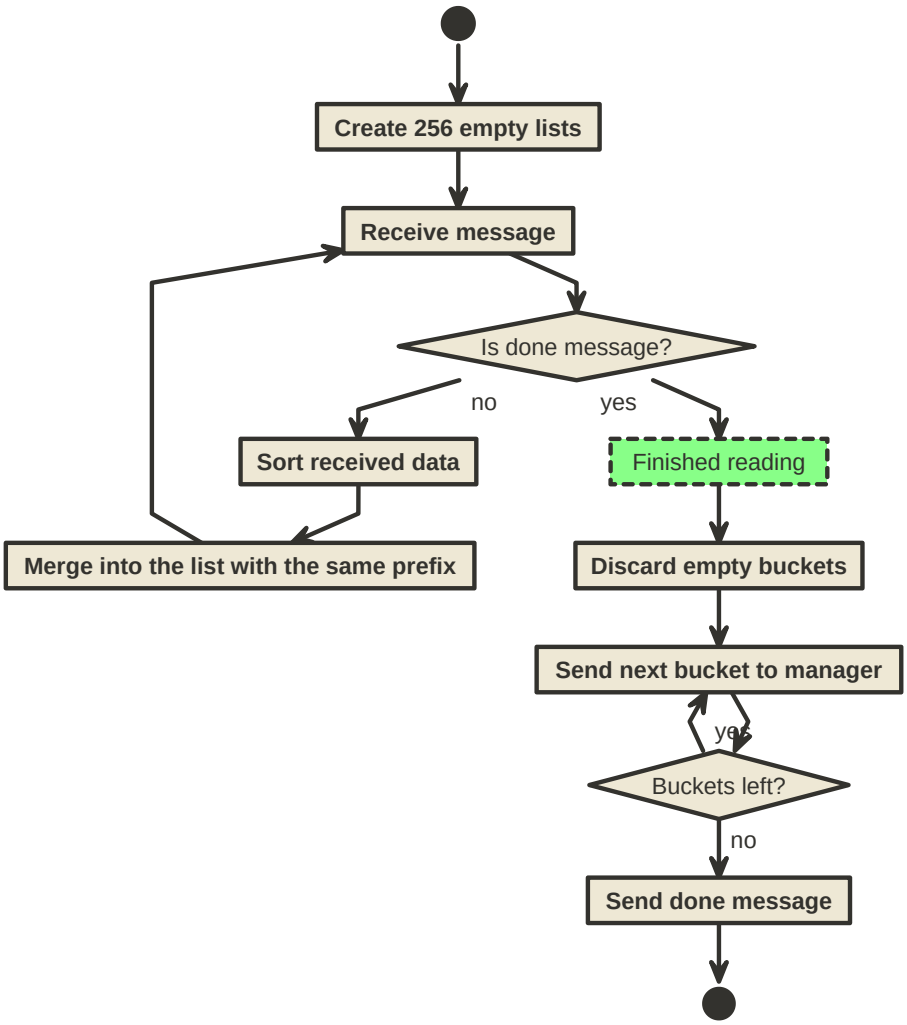

Figure 2. Worker flow

MEASUREMENTS

We measured performance for datasets sized between 2^10 to 2^30 entries on the Virgo cluster. These datasets were sorted by 4 algorithms:

`radix-sort`
: A single threaded radix sort implementation.

`unstable_sort`
: The rust standard librarys `sort_unstable()` function which uses pattern-defeating quicksort.

`mpi-single`
: Our MPI implementation with a single thread per node.

`mpi-multi`
: Our MPI implementation with 2-4 threades per node, so reading, transmitting, sorting, and writing can happen in parallel.

We tested the MPI implementations with 1 to 16 nodes and 1 to 16 tasks per node (but at least 2 tasks). The two non-MPI implementations were tested with one node/one task. First the input file is copied to the `/tmp` directory on the manager node. The sorting algrithm then uses that file as input to avoid lustre bottlenecks. The output file is also written to the `/tmp` directory on the manager node. The measurement does not include coping the input file to the `/tmp` directory or copying the output file from the `/tmp` directory. The measurement does include reading and writing of the files from the `/tmp` directory. All measurements were made from inside our application. We also measured individual times for reading, transmitting, sorting, and writing to indentify bottlenecks. We tried to measure each implementation for each applicable combination of nodes and tasks per node 16 times. We bundled four measurements for the same configuration into one slurm batch. In reality we got fewer measurements, as they sometimes fail.

NON DISTRIBUTED PERFORMANCE

We want to select a non-distributed implementation to use as a baseline when comparing the distributed implementations. Figure 3 shows the total runtime of the two tested non-distributed implementations for different dataset sizes. It shows that `sort-unstable` is faster than `radix-sort` for smaller datasets. It's measurements are also more predictable, as the curve does not have a weird bump at 2^18 bytes. Figure 4 shows that the bump is probably not a measurement error, as the difference is in the sorting step and not while reading the input or writing the result.

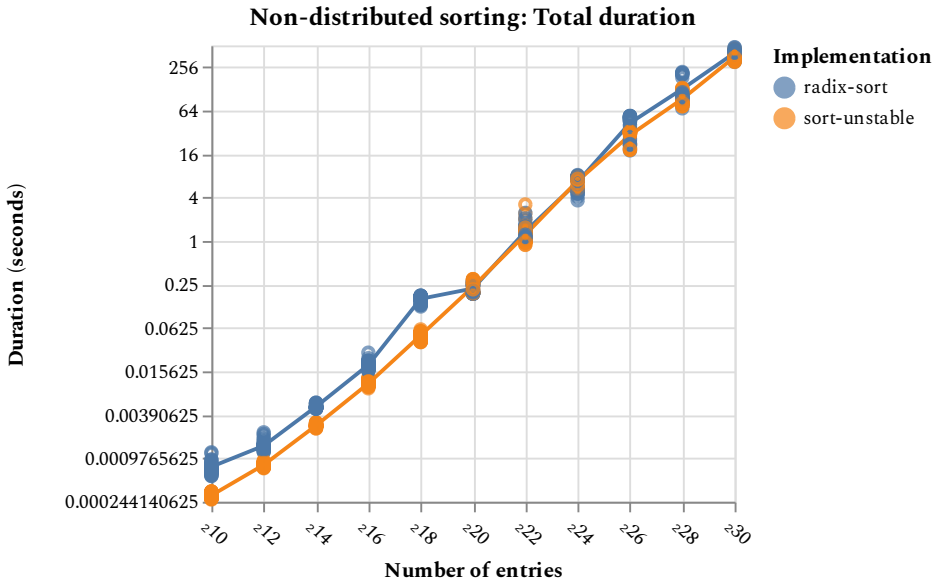

Figure 3. Total runtime comparison of the non-distributed implementations

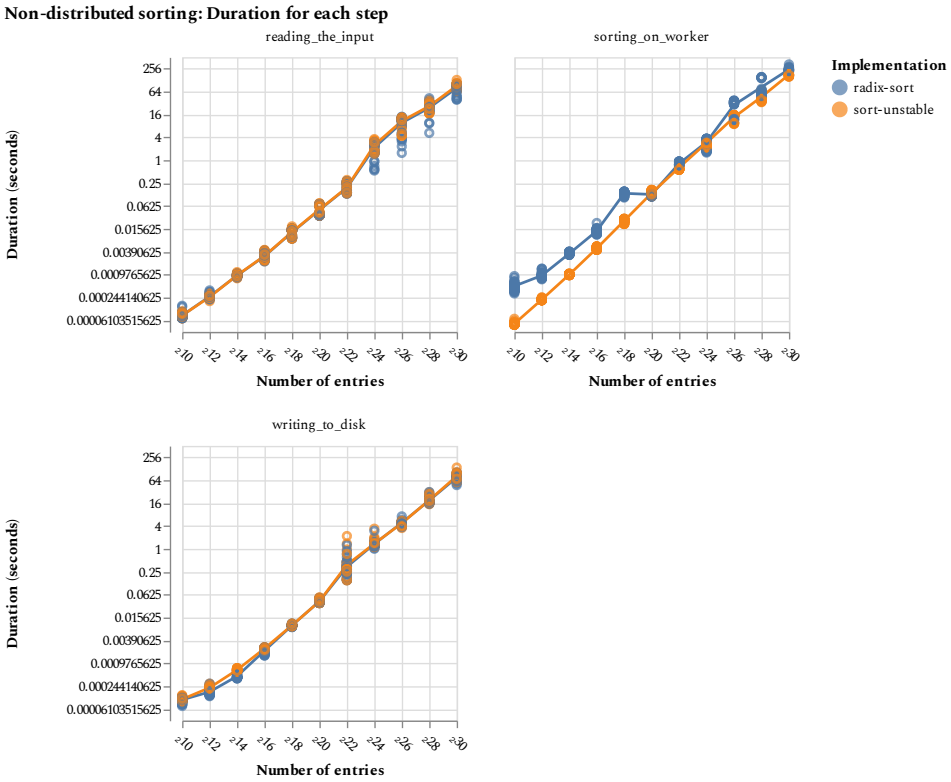

Figure 4. Runtime comparison of the non-distributed implementations

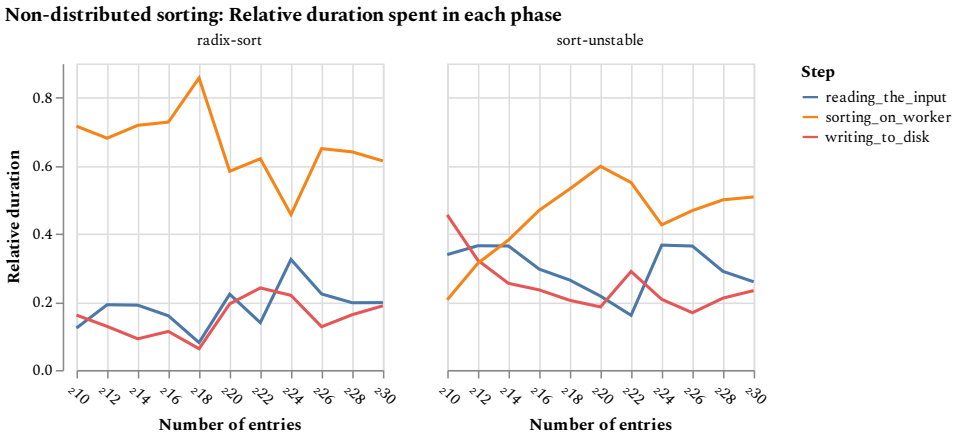

Figure 5. Relative step comparison of the non-distributed implementations

We will use the `sort-unstable` implementation as a baseline for the distributed implementations, because it is faster and has a more predictable runtime.

DISTRIBUTED PERFORMANCE

Figure 6 shows the relative speedup of the distributed implementation with different numbers of tasks compared to `sort-unstable`. Each line represents a implementation running with a given number of ranks. Every rank runs on the same machine.

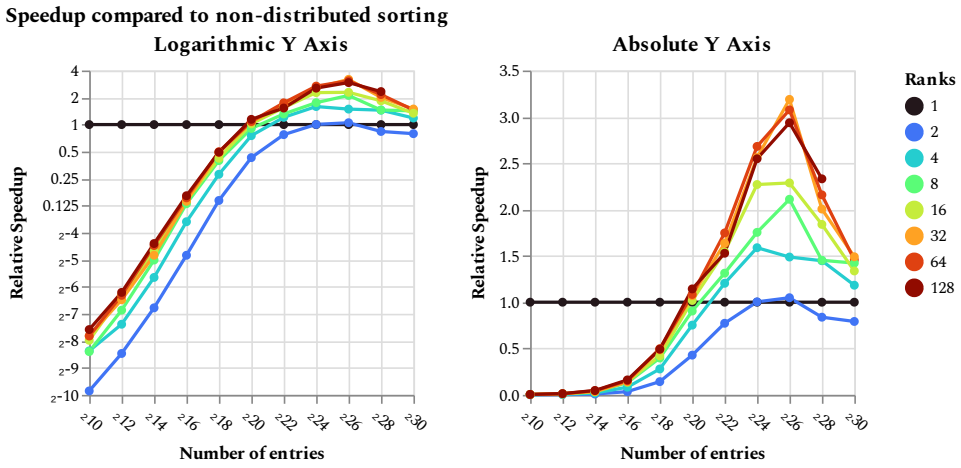

Figure 6. Speedup compared to non-distributed sorting

It shows that the distributed implementation is slower than the non-distributed implementation for small problems (n ⇐ 2^20 entries). The best relative speedup is achieved for problems with a size of 2^24 , after that the relative speedup gets lower again.

When only two tasks are used, there is no speedup. With 2^24 entries the speedup is exactly 1, which means that the distributed implementation is as fast as the non-distributed implementation. I would have expected it to be lower, as we only have one worker that is actually sorting in that case. The advantage we get by bucketing the data and only sorting small chunks, seems to be equal to the overhead of distributing it into buckets and sending the data over the network.

When using 4 or 8 tasks the speedup is 2 for 2^24 entries. For 16, 32, and 64 tasks the speedup is 3 for 2^24 entries. It looks like the maximum speedup is achieved when using 16 tasks, after that we get diminishing returns.

The efficiency behaves similar to the speedup. The best effieciency is reached for problem sizes around 2^26. The more ranks we use, the lower the efficiency gets. This is to be expected, because of the communication overhead. The efficiency is always lower than 1, which means that the distributed implementation is always slower than the non-distributed implementation.

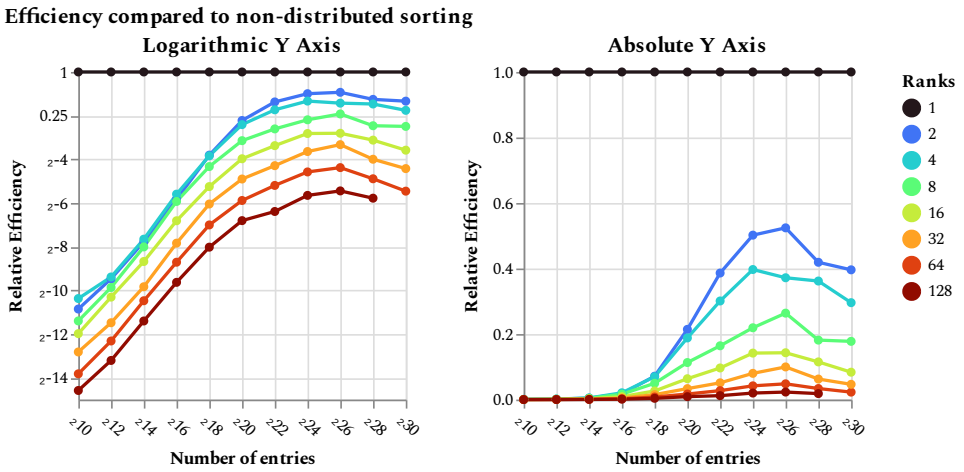

Figure 7. Efficiency compared to non-distributed sorting

Figure 8 shows how the runtime is distributed over the different steps of the algorithm. As most of the sorting is done during the receiving on the worker nodes, the sending step on the manager node scales with the number of tasks, because it measures the time until all tasks received all data. As described in [distributed-implementation] the worker nodes alternate between receiving a

bit of data and sorting it, which is measured seperatly. Because of this the sum of the sorting and receiving step on the workers roughly add up to the sum of the reading, bucketing and sending step on the manager node.

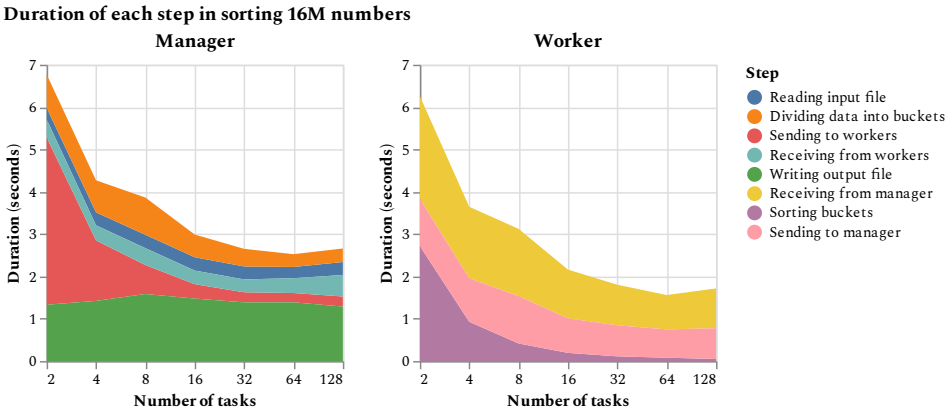

Figure 8. Duration of each step when sorting 16M numbers

Figure 9 sums up the runtime of all tasks and shows how much is spend doing actual sorting and how much is spend doing I/O or waiting for I/O. We can see that most of the time is spend doing I/O operations. The more tasks we add, the more disbalanced the relation between I/O and sorting.

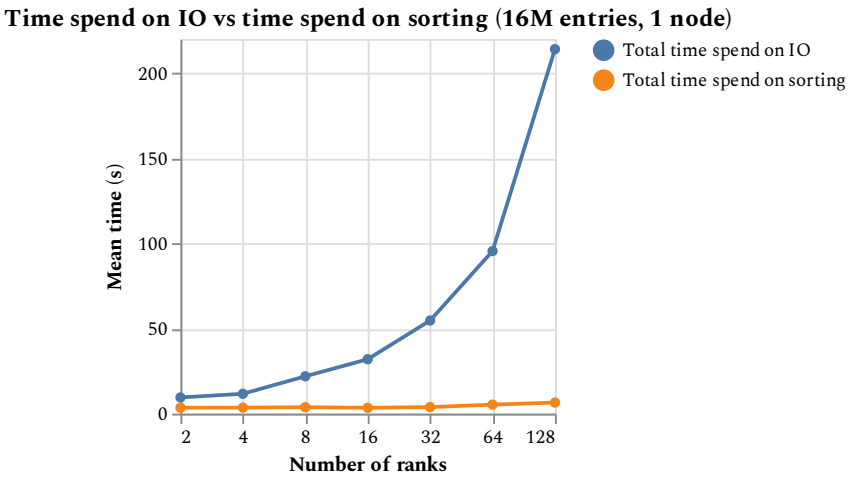

Figure 9. Total time spend on I/O vs total time spend on sorting

This demonstrates that our main bottleneck is distributing the data among all nodes. We tried a few different combinations of nodes, tasks, and problem sizes and it always is the case that the I/O on the manager node is the main bottleneck. We tried to improve the performance by using a different MPI function to send the data, but we did not manage to improve the performance

significantly. We also learned that MPI has a limit of 2GB per buffer, which can be problematic when working with large datasets, like 100GB of sorting data.

RESULTS

The main bottleneck of our implementation is the communication between the manager node and the other nodes. For some specific combinations of problem size and number of nodes we can achieve a speedup of up to 4 compared to the sequential implementation but most of the time the speedup is equal to or lower than 1.

Rust MPI is ok, but not great. It is not very ergonomic as it mostly just wraps C calls. The documentation is acceptable. One of the major pitfalls we encountered was with the `receive_vec` function which is a wrapper around `MPI_recv`. It puts the data directly in a heap allocated vector. It is quite slow for large buffers, as it does not know the size of the data it is receiving in advance, so it has to grow the Vec repeatedly. We ended up using `receive_into` a buffer instead, which is a bit more verbose, but much faster.

The posix fadvise API is cool and can be used to tell the OS that we are going to read a file sequentially and only once. This can improve performance by a tiny bit, as the kernel can prefetch more the data. Memadvise is probably also cool, but we did not have time to try it out.

Lustre can be quite slow, probably due to congestion.