Distributed Sort

Practical Assignment Distributed Systems
Group Three

Ralph Erkamps - Henley Ding - Emmanouil Chalkiadakis

Free University Amsterdam
The Netherlands
December 17, 2020

Abstract

In this report we propose a design for a distributed sorting system together with an analysis of the system. The system Uses a two phase approach to sorting large quantities of data. This is implemented using two types of nodes. Coordinator nodes which manage the system and sorter nodes which process the data. Properties of the system that we explore are the performance, weak and strong scaling and fault-tolerance. Results show that the system scales when increasing the amount of nodes. However the system performs worse than other distributed sorting systems because of flaws in the design.

Introduction

Sorting is an important field of study within computer science. A large quantity of tasks, like search engines or scientific applications, require some form of ordering data to function. Due to the increase in quantity of data, on which those applications work, sorting has become an increasingly complex problem. However, since sorting is one of the largest bottlenecks in large-scale systems, efficient sorting algorithms have become progressively more important [3].
In this report we present the design of our distributed sorting algorithm. Besides the design of the system we also pay attention to the scaling properties of our system alongside performance and fault tolerance. These properties are evaluated with help of several experiments. Through analysis of these results we try to locate the bottlenecks in our implementation. We conclude the paper with a discussion about the strengths and weaknesses present in the design of our sorting system. In a future work section we propose improvements based on the results from our experiments and comparisons with other distributed sorting systems.

Background

The distributed sorting algorithm which we implemented is a system which is capable of sorting large quantities of data. The data our system sorts are large files consisting of many 100 byte tuples of a ten byte key and a 90 byte value. The sorting is done on the key elements. The system is required to be able to sort files larger than fits in the main memory. For this we used a partition then merge approach.
A important requirement that is demanded from the system is scalability. When the workload the system needs to handle increases, there should be no problems applying our distributed sorting algorithm on the data. Using the same amount of resources the system should take longer to sort, but not fail. Besides scaling the data, the system should also be able to be scaled to perform more efficiently when more resources become available for the system.

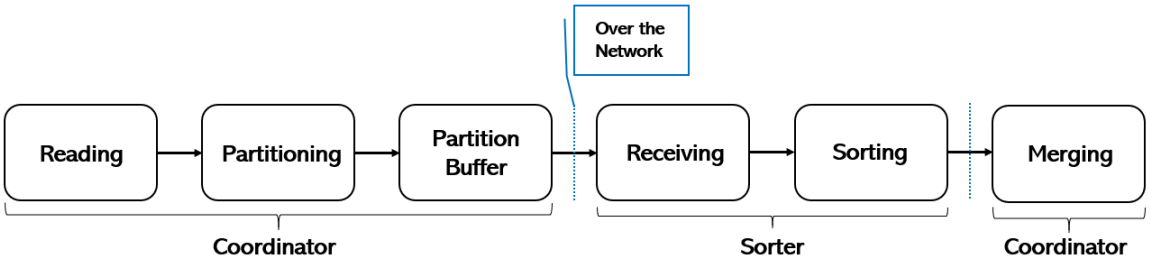

Figure 1: A overview of the system using a directed graph.

System design

Our sorting algorithm is a distributed, pipelined system. In this section we introduce our design and motivate the decisions we made for each of the pipeline components which make up our system.

System overview

Figure one shows the different stages which occur in our sorting algorithm in a directed graph. These stages can be divided into two groups based on the functionality they serve. We divide our system into a partitioning phase and a sorting phase. In the coming subsections we will discuss these phases in more detail.
Another distinction we make in our system is between two different types of nodes. The first node type is the coordinator node. The system has exactly one coordinator node which is decided on system startup. When the coordinator node has been determined it is provided two parameters. The first parameter is the data which needs to be sorted. This has as a consequence that the data should be available on the coordinator node itself. The coordinator is tasked with partitioning the data into ordered subsets and then distributing these among sorter nodes. These sorter nodes are the second type of node that our system uses. Sorter nodes receive partitions of data which fit in main memory and sort it. The second parameter which the coordinator requires is an integer which determines the amount of sorter nodes the system should use. When the coordinator node has been initialised it starts sending out a beacon signal to possible sorter nodes. On the other side when sorter nodes are initialised they start listening for this beacon signal. When they receive this beacon from the coordinator they send a message back and the nodes establish a connection. If the coordinator has found the required amount of sorter nodes it begins the partitioning phase.

Partitioning phase

The partitioning phase is the phase in which the data is partitioned and distributed over the sorter nodes. This corresponds with the first four nodes in the directed graph in figure one. The partitioning phase occurs mostly on the coordinator node.

Reading

he first step in the partitioning phase is the reading of the data. Since the system is required to work with datasets which cannot be stored in main-memory reading the file is done using a reader-buffer. This reader-buffer is a segment of the main memory dedicated to storing the entries of the data-file. The reading of the file is done by repeatedly loading entries from the file on the disk to the reader-buffer in the main memory. After data has been loaded into the reader-buffer it is sent to the partitioning mechanism.

Partitioning

We want the data from the reader-buffer to be partitioned over the sorter-nodes. For each data-element it needs to be determined on which node it will be sorted. This is done by passing each element through a function which maps the key of the data entry to a partition.
An important requirement of this mapping function is that it distributes the data-entries evenly over all partitions. If the workload on one sorter node is significantly larger than the others then that sorter node will require longer to sort. The sorter nodes which are already done sorting then won't contribute to finishing the workload, causing inefficient usage of resources.
To determine to which partition a data entry belongs we use a dictionary. This dictionary maps the keys from the data entries to their corresponding partition. While dictionaries are fast they come with a speed/memory tradeoff. It is unfeasible to create a dictionary which maps each possible key to a partition since the key space is too large. To solve this problem we only look at the first byte of the key when determining the partition. This way our dictionary only needs to map the first character of a key to a partition which significantly reduces memory required for the dictionary. As a tradeoff however our partitions are less precise. This problem becomes bigger if the amount of sorter nodes increases. For example, if we have 255 sorter nodes and the first byte can take 256 values, then that means, if we want to evenly distribute these 256 values over the 255 nodes, that one node has two values mapped to it. This causes it to have twice the workload of other nodes. To solve this problem more bytes can be used to partition when larger amount of sorter nodes are used.
When a data entry is assigned to a partition, it is sent to the partition buffer. The partition buffer is a buffer which contains other buffers. One buffer for each partition. When a data entry is assigned to a partition that value is stored in the corresponding buffer for that partition in the partition buffer. If a buffer for a partition is full then it is sent over the network to the sorter node which

sorts that partition. Finally when all data entries from the data file have been partitioned, the remaining elements in this buffer are sent. Together with the final elements a signal is sent from the coordinator nodes to all sorter nodes indicating that the sorting phase can begin.

Sorting phase

After the sorter nodes receive the signal that indicates that all data entries have been partitioned the sorting phase starts. The sorting phase corresponds to the last two nodes in figure one. A sorter node loads the data it received from the coordinator node into its main memory. Since the data fits into the main memory sorting it becomes a simple problem. Multiple sorting backends have been implemented, the default of which is radixsort. The result is written to a file on the disk if the setting has been specified. Upon completion this resulting sorted file is sent back to the coordinator node.
The coordinator node receives the sorted data from the sorter nodes. Since the files received are ordered and sorted they can be merged into a final sorted file. This is done by appending the received files in order to the final file.

Monitoring

The coordinator constantly monitors that everything runs smoothly. This is done in a hybrid polling and event driven manner. Should anything happen, a watchdog is woken which detects errors and handles them by performing the corresponding fixing action. Moreover, it is guaranteed that the watchdog will wake again in no longer than the configured interval since it last run, even if no error has actively woken it. This core subsystem has unnoticeable performance penalty on the host.

Fault tolerance

Each sorter node has a kill switch and a timer which is reset by packets from the coordinator. If the configured time passes, or any other error occurs, like broken connection, they reset and enter their default state of waiting for beacon. On the the other side, the coordinator monitors the time a node has not sent status update through the watchdog. Initially a soft timeout occurs, where the coordinator queries the node at set intervals. Then, if the node does not respond and the time exceeds the hard timeout threshold, the coordinator restarts the beacon, listens for connections and replaces the faulty node(s). As soon as the faulty node(s) is replaced, the data is partitioned again, and the partition(s) is sent to the corresponding node(s). If multiple nodes fail within the same period of time, they are replaced in batch, so the partitioning is done only once. If an existing node has already finished, its state is reset and is waiting for beacons, it will respond. To avoid unnecessary CPU utilisation, each partition could be written in a separate file the first time it is used at the cost of disk space, in case of future reuse.

Experimental Results

Experimental Setup

Our experiments were conducted on the DAS-5 supercomputer [1]. For each node in our system we used a standard computing node on the DAS. These computing nodes on the DAS-5 have a dual 8-core 2.4 GHz (Intel Haswell E5-2630-v3) CPU configuration and 64 GB memory. We used the InfiniBand network, which offers low latency and throughputs up to 48 Gbit/s. The data was generated using the program gensort [2], which is able to create specific workloads consisting of key value tuples for different sizes. The maximum size of data we could experiment on was 15 million entries because of storage restrictions on the DAS. The code was executed using python 3.4.5, no external libraries were used.

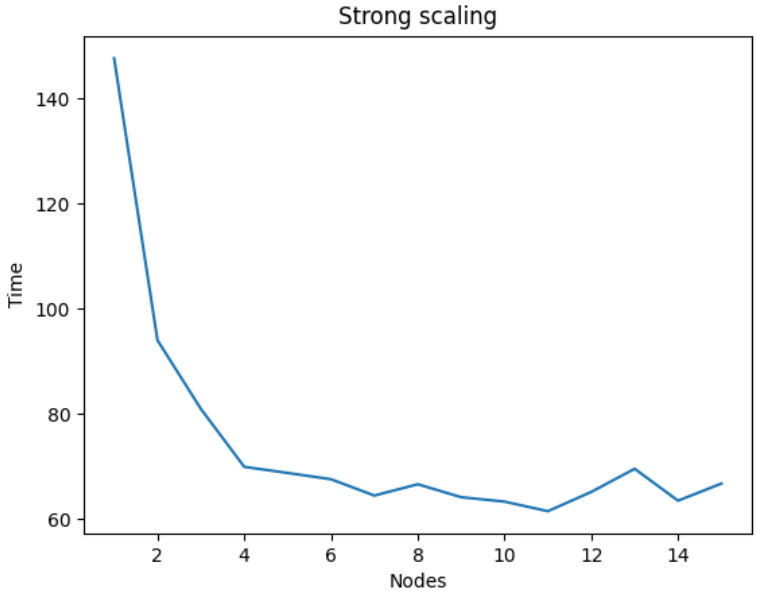

Figure 2: The time in seconds required for our system to sort ten million entries for different amounts of sorter nodes.

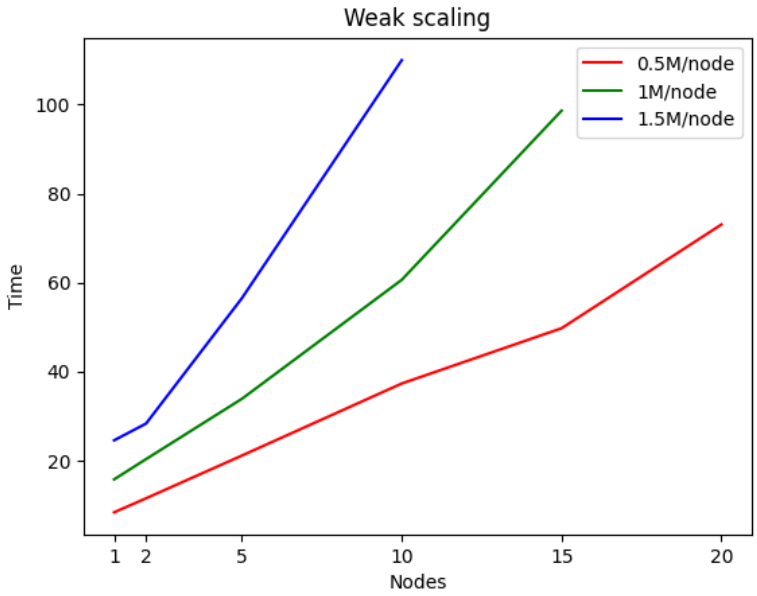

Figure 3: The time in seconds required for our system to sort different datasets for different amounts of sorter nodes while keeping the work done per node the same.

Experiments

Strong scaling

To test the strong scaling of our system we measured the time required to sort a given workload for varying amounts of sorter nodes. The size of the workload used is 10 million entries. The result are shown in figure two. We observe that adding resources to the system improves sorting time up till a certain point. This improvement continues until a threshold is reached and no more speedup is achieved. The best performance was observed using 11 sorter node with a time of 61.53 seconds. The worst performance was measured for one sorter node with a speed of 147.59 seconds. The maximal speedup measured was 2.4 .

Weak scaling

To test the weak scaling of our system we measured the time required to sort a given workload for varying amounts of sorter nodes. The size of the workload is decided such that the amount of work on each processor remains the same. We see a increasing linear pattern in our graph when the amount nodes increases. This can be explained by the fact that the coordinator nodes needs longer for the partitioning phase when the amount of data increases.

Fault Tolerance

To test the fault tolerance of our system, we started a 10 million entries on 10 nodes experiment normally. Some time later, we killed two nodes, having already spun up some extra, which took their place immediately. The results were the following: The nodes were killed 7 seconds after the connection had been established. The experiment was completed in 91.5 seconds. To compare, we run the experiment again using the same dataset without killing the nodes. The program finished in 64 seconds.

Bottleneck

During our experiments, we observed that our program was partly sequential, which appeared to be the bottleneck. The system reports the time difference between state change updates from the sorters. Increasing nodes seems to lead into slight network congestion, and a small increase in the data sending phase across all nodes. For the 10 million entries experiment, one node spends 23 seconds on network, and 92 seconds sorting. For the same experiment, 10 nodes spend an average of 24 seconds each on network, and only an average of 9 seconds each sorting. The data returning phase was always faster, averaging 18 seconds per node.

Discussion

For testing purposes we haven't used datasets larger than 15 million entries. While our system works for these data sizes it is incredibly inefficient. This is because the data can still be stored completely in the main memory and sorted there without the additional overhead that occurs by using our system. However for datasets of larger magnitudes, which can't fit in a machines main memory, then our system will be a better alternative. This is under the assumption that there are enough resources such that the data can be stored in the combined main memory of all the sorter nodes.

In our system we have chosen for a single coordinator node. This decision causes our design to have certain advantages and disadvantages over other designs which don't use a single coordinator. An advantage of the single coordinator approach is that there is no need for consensus between the different sorter nodes. All decisions are made solely by the coordinator node. For example, which partition is assigned to which sorter node. Another advantage is that the amount of connections is minimal. Sorter nodes only need one connection, between the coordinator and itself. Since the amount of connections is minimal, the chance that there is connection failure is smaller than other designs which require more connections to function.
A single coordinator also brings disadvantages. Since the coordinator node is essential for functioning there is a central point of failure. When the coordinator node fails, the whole system fails. Another disadvantage in our system is that the coordinator node itself is unable to sort data. The coordinator is waiting on the sorter nodes while not working on the workload. Another problem is that the sorter nodes need to wait on the coordinator node before they can start sorting. They can only start if they have received a signal from the coordinator that all data has been sent. During this waiting time on the signal no work is done on the sorter nodes.

Future work

Our design has many sub-optimal components. In this section we will discuss how we would improve our design.
As discussed above, the single coordinator implementation comes with many disadvantages that we believe far outweigh the advantages. There is a single point of failure, the sorters are not working during partitioning and the coordinator is not working during sorting. We believe that these weaknesses can be resolved by using a decentralised design. We propose a design in which each node can perform the sorting and partitioning, therefore discarding the notion of sorter and coordinator nodes. Just like our current design the system uses the two phase approach. However now instead all nodes can work simultaneously on the workload. Of course the system would require extra functionality to make this happen. There is no central coordinator which assigns partitions for the sorter nodes, therefore the system requires a consensus algorithm which agrees on a parti-

tion for each node. After each node has agreed on a partition the data elements can be sent to the correct partitions. Since every node now is aware of the partition distribution over the other nodes they can all contribute to the partitioning of the data. After the partitioning they simply sort the data which can then be merged into a final output file.

Conclusion

In this report we proposed a design for a two phase distributed sorting system. Experiments were performed testing the scalability and performance using the DAS-5 supercomputer. Our system was able to sort datasets correctly. We were unable to test for datasets of larger magnitudes than 15 million entries because of storage limitations on the DAS. Analysis of the sorting on the DAS showed that Our system can achieve a maximum speedup of 2.4 through strong scaling. For weak scaling we saw a linear increase in time. Fault tolerance successfully prevents the system from failure when sorter nodes fail, this recovery however increases the sorting time. A bottleneck was observed in the sequential part of our program which occurs on the coordinator node. We conclude that a design which has no coordinator node might remove this bottleneck.

References

[1] *DAS5*. Dec. 2020. URL: `https://www.cs.vu.nl/das5/`.

[2] *Gensort*. Dec. 2020. URL: `http://www.ordinal.com/gensort.html`.

[3] Alexander Rasmussen et al. "TritonSort: A Balanced Large-Scale Sorting System." In: