Distributed Systems

X_400130

Lab Assignment

Distributed Bucket Sort

On the DAS-5

Authors

Name	Student ID	Email
Joseph Groot Kormelink	2655649	j.g.c.grootkormelink@student.vu.nl
Jakob Kyselica	2593776	j.m.kyselica@student.vu.nl
Noah Voogd	2666711	s.n.voogd@student.vu.nl

Support cast

Name	Role
Animesh Kumar Trivedi	Supervisor

Submitted on

22-12-2019

Table of contents

Abstract 2

Term definitions 2

Introduction 2

Background 3

System Design 3

Input generation 3

Bucket sort 4

Functional requirements 4

Non-functional requirements 5

System overview 5

Implementation 6

Output validation 8

Packet size 8

Experimental results 8

Experimental setup 8

Experiments 9

Discussion 13

Conclusion 14

Appendix A 15

Time sheet 15

Abstract

Sorting values is an extremely common operation in data processing. Nowadays, data processing is often done on distributed systems, as is sorting. In this project we have designed and implemented our own distributed sorting algorithm on the DAS-5, a distributed system accessible to us through the VU Amsterdam. Our algorithm is a distributed form of bucket sort. Although it does not compare to state-of-the-art algorithms fine tuned for specific hardware, it is functional and offers a performance gain when compared to sorting on a single machine.

Term definitions

In this report we use certain terms seemingly interchangeably. The table below describes the definitions we used.

Term	**Definition**
project	Comprises the entirety of what we have worked on, i.e. the design of our sorting algorithm, its implementation in code, its execution on the distributed system (the DAS-5) and the report.
system	The implementation of our sorting algorithm in code and its implementation on the distributed system (the DAS-5).
application	Any code that executes a function.
algorithm	A description of steps to follow. In our case (sorting algorithm) a description of steps to follow to convert an input of random elements into an output of the input elements, but in sorted order.

Introduction

Sorting values is one of the most common operations in data processing. Optimizing this seemingly simple operation would therefore yield great benefits, reducing the runtime of whatever application makes use of it. Now that distributed systems have become commonplace, sorting values in a distributed manner has become part of the sorting-optimization problem. Out of necessity, due to the design of distributed applications, but also because making use of all the resources in a distributed system could potentially lead to further optimization.

In this project we have designed and implemented a distributed sorting algorithm that sorts a list of key-value pairs by key. This is based on the 'Indy' sorting challenge taken from sortbenchmark.org. Here an input generator is supplied, which can generate any number of key-value pairs to sort, along with an output validator, which checks the sorted result for correctness. We made use of these tools in our project.

The website also lists previous winners of the sorting challenge posed there, along with links to papers describing the approaches the winners took to implement there sorting algorithms. We looked at these for inspiration.
As a group we already had basic knowledge of sorting algorithms and parallel programming, which supplied us with a vague notion of how to implement a distributed sorting algorithm.

We decided upon implementing a custom version of bucket sort, due to the nature of the input we want to sort and the characteristics of bucket sort, making it very suitable for parallelization/distribution. The nodes within the distributed system on which our algorithm will run will act as the buckets in our custom bucket sort algorithm.

This report covers the background on our system, what was used to implement and run it, and its requirements. Next we will describe our system design and its features, followed by the experiments that were run to test the system and the results of these experiments. Finally, we will discuss our findings and conclusion.

Background

Our project is based on the sorting challenge posed on sortbenchmark.org. This website aims at inspiring people to come up with new and/or improved sorting algorithms for sorting on distributed systems.
There are two benchmark categories, ‘Daytona’ and ‘Indy’. The category that we are concerned with, Indy, deals with sorting 100-byte records with 10-byte random keys. The Daytona category deals with sorting all kinds of inputs, making it more general purpose.
The website lists multiple benchmarks, which are comprised of sorting algorithms that achieve certain sort rates and data costs.
The website includes a FAQ page where the exact criteria that a sorting algorithm written for the challenge should comply with.

Our system is written in Python, because it makes for fast debugging thanks to its interpreter, we already had experience with it and because the distributed system that we want to run our system on, the DAS5, supports Python.

System Design

Input generation

For the generation of our input elements, we make use of ‘gensort’, an executable supplied by *sortbenchmark.org*. We use ‘gensort’ to generate a list of 100-byte key-value pairs, which consist of a 10 byte *key* followed by a 90 byte *value*. The input is to be sorted by *key*. The *value* of the item is not relevant to the sorting, and is not used in our algorithm. However, each value stays associated with its respective key throughout the sorting process, as prescribed by the benchmark rules. The *key* part consists of a 10 character ASCII string (10 bytes). The character range of the characters in the key starts at ASCII value 32 (the ‘space’ character) and ends at ASCII value 126 (the ‘~’), thus giving us a range length of 95.

Bucket sort

Our system is essentially a distributed implementation of bucket sort. Bucket sort works by dividing the input into 'buckets' such that, when the buckets are individually sorted and concatenated, the result is the original input in order. This is achieved by partitioning the input such that every item in one bucket is smaller than every other item in the next bucket, regardless of intra-bucket order. This algorithm can be very effective in parallel or distributed situations, because the initial partitioning into buckets requires no actual comparisons between items, and can be performed in O(n). This partitioning then allows the buckets to be sorted completely independently of each other, which results in an *embarrassingly parallel* workload. However, since each bucket is sorted independently, the performance of a parallel bucket sort greatly depends on how the input is distributed across the buckets. If one bucket ends up with far more data than the other buckets, it will take much longer to sort, thereby leaving the other processes idle and defeating the purpose of the algorithm. Ideally, each bucket receives an amount of data to sort, such that each bucket is finished sorting at roughly the same time. In a homogenous system, this means all the buckets receive more or less the same amount of data. In a heterogenous system the performance of the individual buckets may be taken into account.
In order to assign the input elements to buckets effectively, it helps to have some knowledge of the input, such as the size of the input or the lowest/highest input value.

For our project, we have the advantage of knowing that the input generated by 'gensort' is **uniformly distributed**. This means that once the input is sorted by key, all the keys will be at equal 'distance' from each other, spread out over their total range. Knowing this, the total range of keys can simply be evenly divided among the number of buckets. Then, upon distributing the input elements across the buckets based on which part of the total range their key falls in, every bucket will be left with roughly the same amount of input elements.
Furthermore, because of the uniform distribution of the input, when assigning an input element to a bucket, we do not need to know about the existence of any other elements, as long as we know the range of the keys and the number of buckets. This allows for the parallelization/distribution of the bucket-assigning operation, which could be beneficial.

Functional requirements

Our system has the following functional requirements:

- The output is correctly sorted function of the input
    - The output is written to a file and can be validated
- The sorting must be done simultaneously on multiple nodes
- A shared service (single machine) can only be used for reading the initial input and writing the final output
- Input must be able to exceed the memory of a single nodes (but not combined memory of nodes)
- OS-agnostic (as long as it can run a Python script)

Non-functional requirements

Our system has the following non-functional requirements:

- It should be as fast as possible
- It should scale, i.e. using more nodes to sort the input should yield an improvement in performance

System overview

Our system works in the following steps (see figure 1):

1. The input is generated by a single node (the master node) using ‘gensort’.
2. The input is distributed among all available nodes such that all nodes get an equal portion of the input.
3. Each node assigns its input elements to a ‘bucket’ that corresponds with a node in the system.
4. Each node sends the input elements to their corresponding nodes (keeping those that are for itself).
5. Each node individually sorts its input elements by key.
6. Each node sends its sorted input list to the master node.
7. The master node concatenates all the received input lists and writes the result to multiple files.

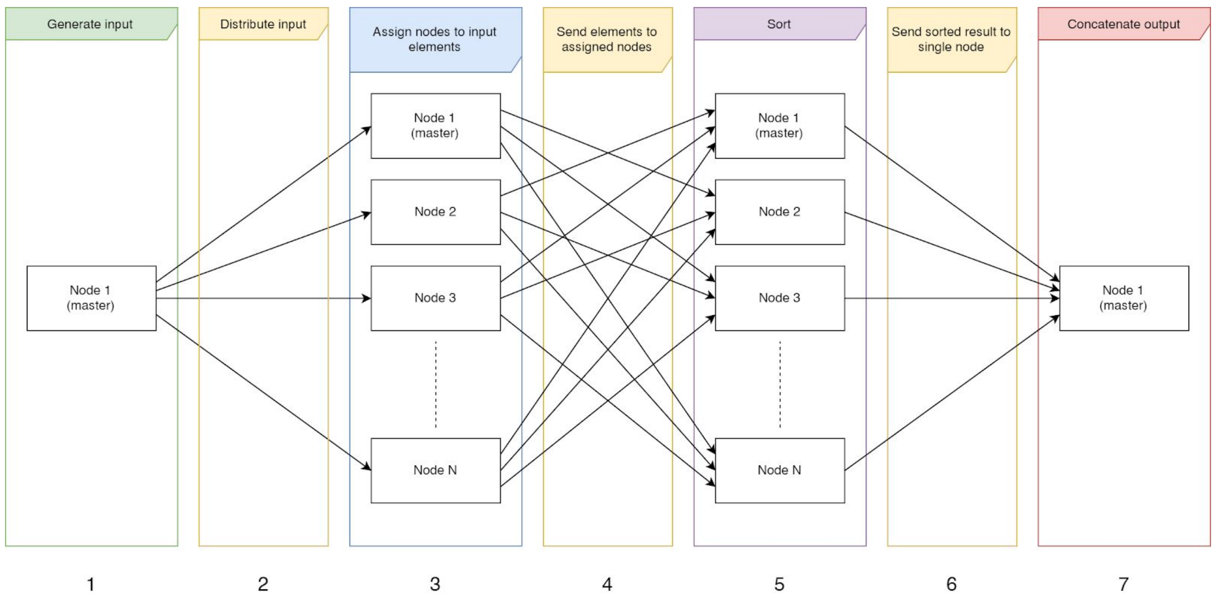


Figure 1: Distributed Bucket Sort

Note that although a single node runs ‘gensort’ to generate our input, the input is not fully loaded into the master node. When sending a portion of the input to fellow nodes, the master node reads the input per line from the memory as to not have to store the input in its entirety in its own memory. After redistribution, a new chunk of memory is read, so we can surpass RAM memory limits on the input data.

As described, the actual distribution into buckets happens in parallel on all nodes. A different way to approach the bucket distribution would be to have the main node already partition the input into buckets before sending it to the nodes, which would remove the need for inter-node communication. We chose the parallel method, because of its positive effect on scalability. With a really large input, the master node partitioning the buckets could lead to a bottleneck, and a single weak point in the system. With our method, the master node performs (nearly) no work on the data, delegating this to the nodes. The only bottleneck would be the IO speed of the master node.

For the actual sorting of the data inside the nodes, we simply use the default sort function within Python, which implements ‘Timsort’. Timsort has a worst-case performance of O(n log n), a best-case performance of O(n) and an average performance of O(n log n).
During the orientation stage of the project, we implemented and tested various parallel sorting algorithms that would make use of multiple cores on the nodes, but we failed to derive any (significant) performance benefits from this. Therefore we opted to simply run a single-threaded sorting algorithm, taking into account the fact that the focus of this project is on the distributed architecture rather than really well optimised single-machine sorting algorithms.

Since each node in the system only ever receives a fraction of the total input data, the size of the input data can exceed the memory (and storage) of a single node. This property scales linearly, meaning that more nodes will always increase the amount of data that can be sorted.

Implementation

The system is implemented in Python 3 using TCP sockets as the method of communication. As described above, a master-worker architecture is used. For the master to be able to discover nodes and their ‘location’, a very simple naming scheme and handshake process are used. Figure 2 shows this procedure.

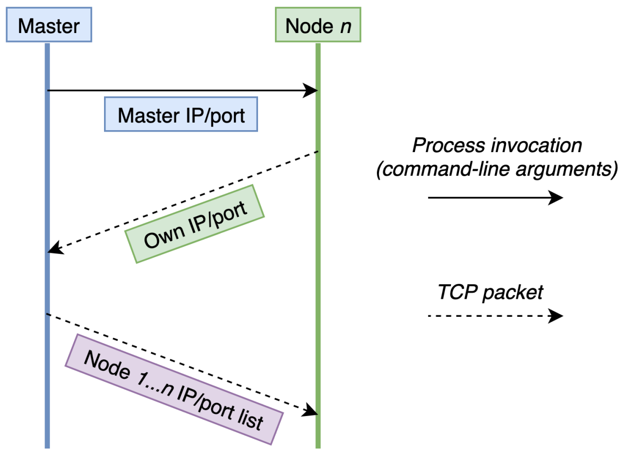

Master
Node *n*
Master IP/port
Own IP/port
Node 1...*n* IP/port list
Process invocation (command-line arguments)
TCP packet

Figure 2: Master-worker handshake procedure

The master node invokes the worker process on remote nodes using its own connection endpoint address as command line arguments. The nodes then start by creating their own socket for inter-node communication, and sending the address for that socket back to the master. The master waits until it hears back from all nodes that were invoked, after which it compiles all the socket addresses into a list that is sent to each node. For the master to be able to identify the different nodes, a simple index is used. The nodes do not need to know their own name, so this name is only kept at the master. The result of this handshake and naming process is that the master now has a connection and identifier for each node, and each node has a complete list of the nodes it can connect to.

Once the handshake is completed, the data is evenly sent to the nodes, after which they begin a 5-stage process. These stages are listed below:

1. Reading
2. Redistribution
3. Gathering
4. Sorting
5. Returning

We will briefly describe the implementation of each stage of the process.

1. Reading

In the reading step the data is gathered from the master. All data is received in chunks which are subsequently split into individual items, each with a single key and payload. As the nodes do not know the amount of data they will be receiving beforehand, they wait for the master to end the initial data transmission with a **"[STOP]"** packet.

2. Redistribution

Redistribution is the second phase of the bucket sort algorithm. For every item in the node's data, a hash based on the *key* is calculated. This hash is used to map the item to its corresponding node. Once the item is mapped to a node, it is added to the outbound buffer for that node. Data is first stored in this buffer as opposed to directly sending it to the receiving node in order to reduce packet overhead and make optimal use of the infiniband connection.

Our implemented hash function is very simple. The keys of the items consist of ASCII characters with a value range of 95, starting with an ASCII value of 32 and ending at 126. We divide this range over the number of nodes that will be distributed to, and start by looking at the ASCII value of the first character in the *key* to determine which bucket that item belongs to. For this project, it is unlikely that there will ever be more than 95 'buckets', i.e. nodes, to divide the input into, so in most cases, items can be assigned to a node based on the first character of their key. However, if the data *has* to be distributed to more than 95 nodes, the hash function finds all the nodes that the element could be assigned to based on its first character and then recursively moves onto the next character in the key until it can determine which single node it should be assigned to.

3. Gathering
After redistribution, all data is gathered to the corresponding nodes. This phase ensures that all nodes have received their data correctly by reading from their sockets, and all nodes are synced in the process.

4. Sorting
This phase sorts the data that has been gathered. As described, the standard Python Timsort is used, as this is an efficient implementation with a good best case scenario, and an acceptable worst case scenario.

5. Returning
In this phase all data is returned to the master node. The master node knows the ordering of the nodes, and thus knows which node output should be written to which result file. TCP guarantees the correct ordering of all packets, and thus all data is guaranteed to be in order.
Finally, to be able to accurately assess the time each step of the process takes, profiling is built in to the program. Each step prints its execution time and call-count to the console after completion, allowing for fine-grained analysis of bottlenecks.

Output validation

For the validation of our output file, we make use of ‘valsort’, an executable supplied by *sortbenchmark.org*. Upon running the executable and feeding out our output file we are told whether or not our output file is the original input in sorted order.

Packet size

One of the problems we had was related to the packet size. Packets were sent with 100 bytes of data each. This resulted in extremely poor performance, with a network speed of 1Mbit/s, where the optimum would be around 40Gbit/s. After using *netperf* and our own preliminary testing we have found 100.000 bytes to work best as package size. This has significantly improved our system speed in comparison to the initial 100 bytes packets. Increasing the packet size further no longer had any significant effect on the performance.

Experimental results

Experimental setup

Our distributed experiments were run on the Distributed ASCI Supercomputer (DAS-5), a six-cluster wide-area distributed system. One of the clusters is situated at the Vrije Universiteit Amsterdam, the university being one of the main parties responsible for the creation of the DAS-5. The nodes within the clusters are connected over Ethernet (Gbit/s at the compute nodes, 10 Gbit/s on the head nodes) and over high speed FDR InfiniBand.
See the DAS-5 website for the exact system specifications
(https://www.cs.vu.nl/das5/clusters.shtml)

For the purpose of comparing our distributed system to the performance of our algorithm on a single machine, our single machine experiments were run on a single node in the DAS-5 system.

Experiments

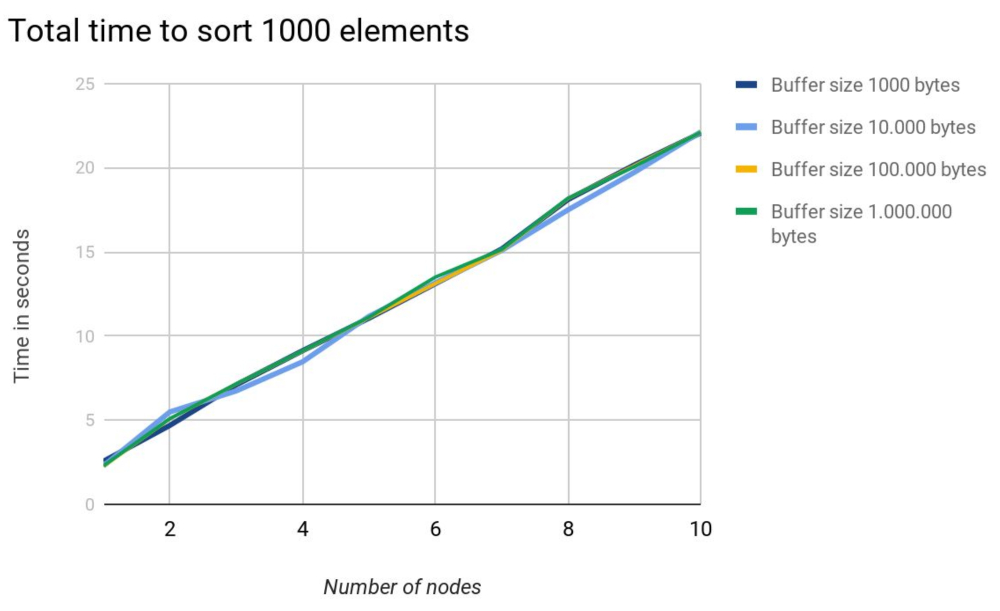
Performance testing

Our algorithm has 2 main parameters:

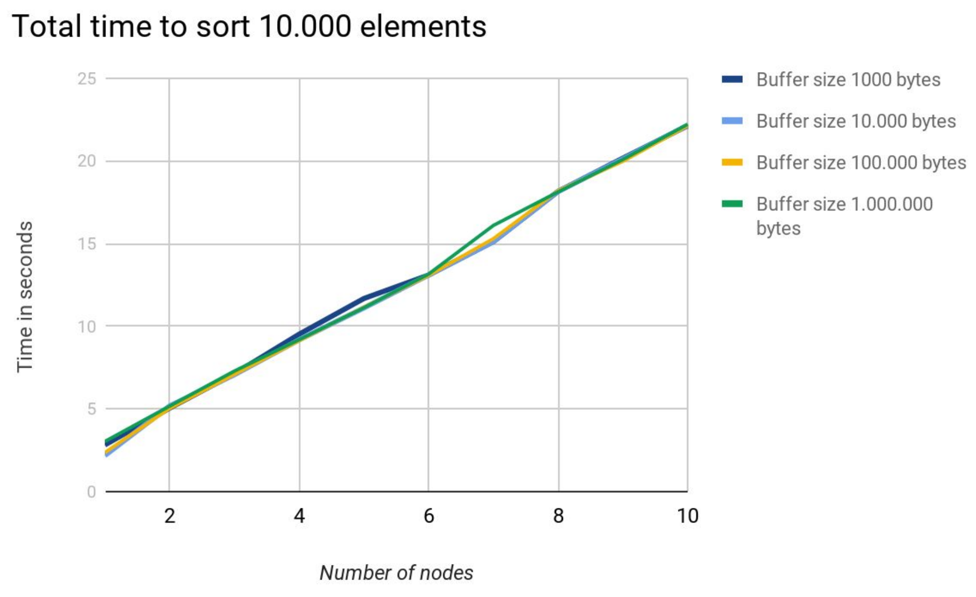
1. The number of nodes that the input data is distributed to
2. The buffer size (the size of the data packages that are sent between nodes)

For our experiments we ended up limiting ourselves to 10 nodes. We are running our experiments on a shared service, the DAS-5, which seemed to prevent us from accessing more than 10 nodes at a time. We were not entirely sure why this was the case, and the system sometimes just appeared unresponsive, so we could not figure out was going wrong. However, for the purpose of our project 10 nodes should be enough to visualize the scaling properties of our algorithm.

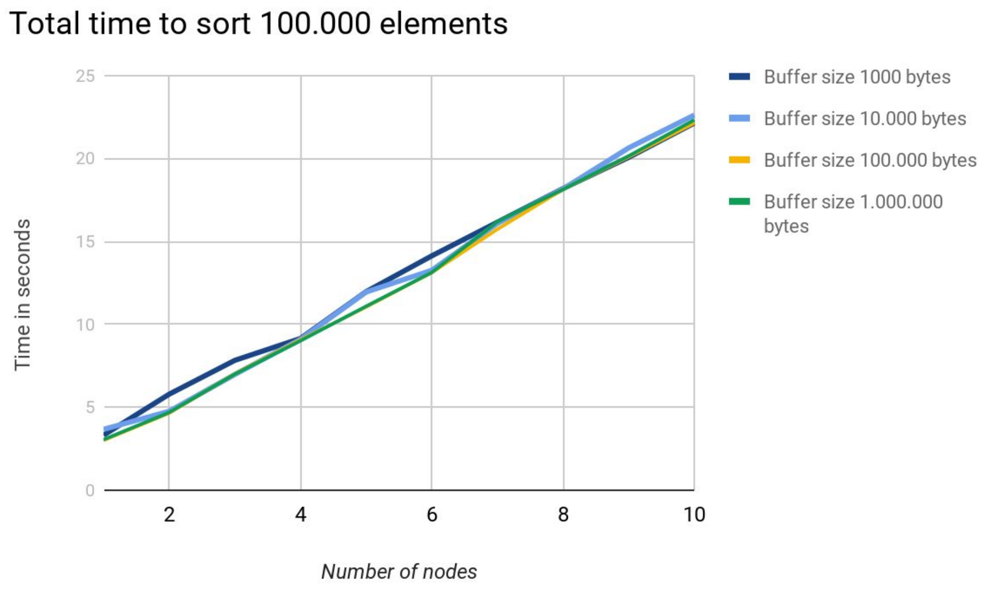
The graphs below show the results of our algorithm being run on respectively 1000, 10.000, 100.000, 1.000.000 and 10.000.000 input elements, using 4 different buffer sizes, on 1 to 10 nodes. The experiments were run 5 times each and the average runtime was plotted onto the graphs.



Graph 1



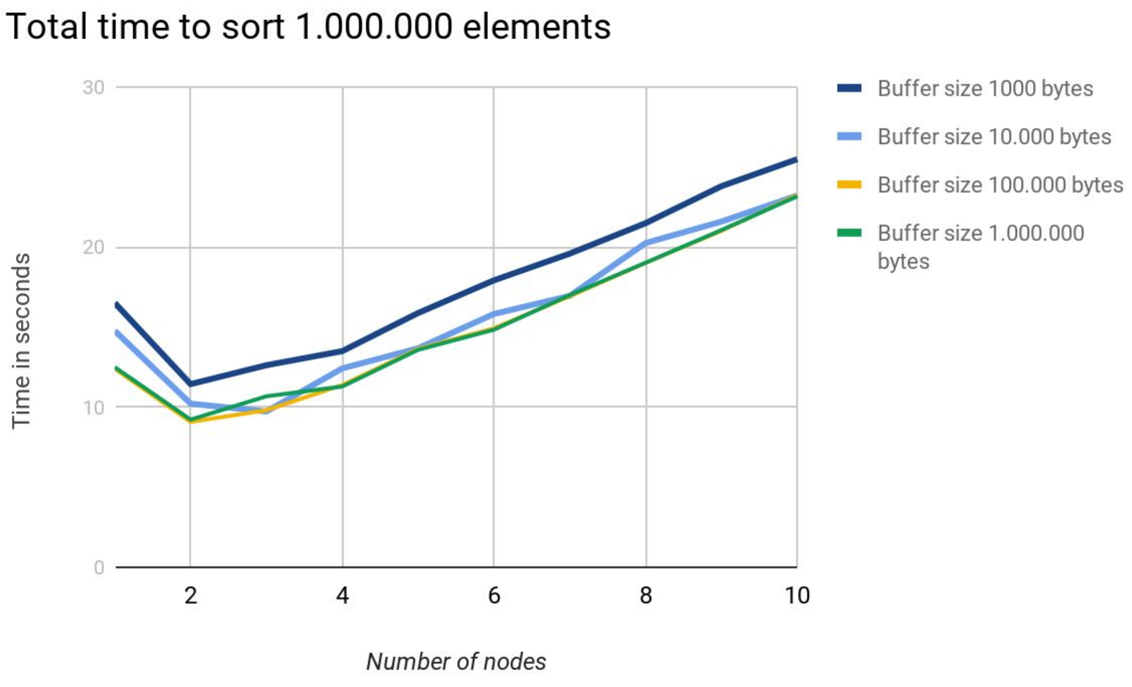
Graph 2



Graph 3

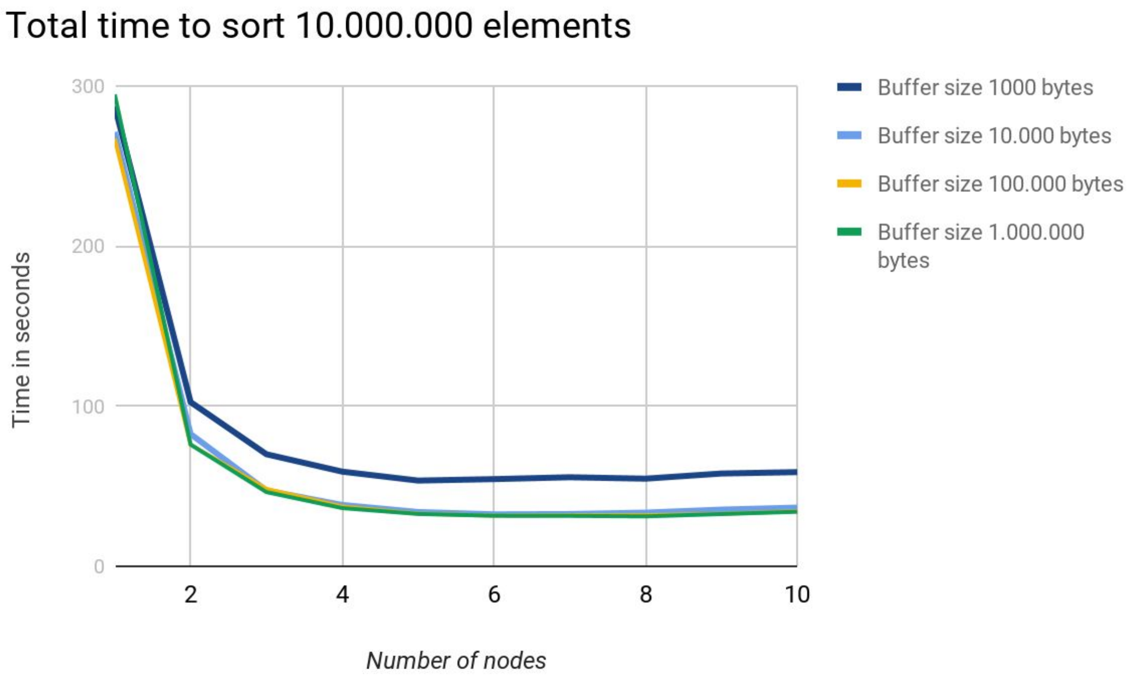
As we can see, for up to 100.000 elements, our algorithm shows no improvement in terms of speed when more nodes are used to sort the data. Sorting on a single node proves the fastest. We believe this is the case because the time it takes to distribute the data over multiple nodes and collecting it again is greater than the sort time of the individual nodes.

The buffer size, too, appears to have no effect on the performance. Sorting using a 1000 byte buffer is roughly as fast as using a buffer of 100.000 bytes.



Graph 4

At 1.000.000 elements we start to see a change. Sorting on 2 nodes proves faster than sorting on a single node, after which the performance once again declines. It appears that at this point, the time to distribute and collect the data starts to balance out the sort time of the individual nodes. At this point we should expect to see a more significant speed up if we increase the input size more.



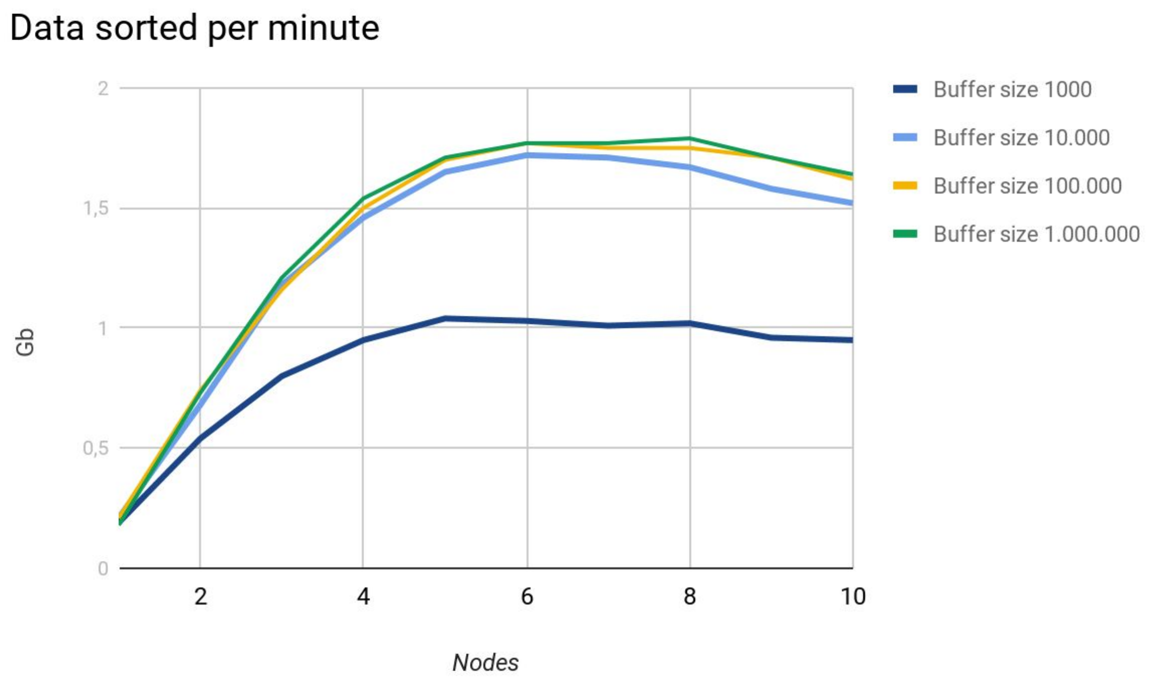
Graph 5

And indeed, at 10.000.000 nodes our algorithm starts to show an enhanced performance that improves when more nodes are added. At 5 nodes the performance no longer improves, but stays roughly the same.

A logical next step would be to increase the input size even more, so we proceeded to sort 100.000.000 elements. However, this resulted in strange behavior from the DAS-5. Probably due to the immense size of the data it now had to process, it would become unresponsive until the connection was manually reestablished. Sometimes the sorting had finished and some kind of result would be displayed, and sometimes it appeared that the process had halted somewhere, leaving us with no output log. This made testing extremely difficult and time consuming.

At last we managed to squeeze a few results out of the system. Sorting 100.000.000 elements on 10 nodes with a buffer size of 100.000 bytes or larger sorted the data in roughly 224 seconds, as opposed to 382 seconds on 6 nodes. This would suggest that upon increasing the input size further, the performance does indeed improve when adding more nodes.

Minute Sort
As described on the sort benchmark website, we can compare our algorithm to other that of other contestants by finding out how much data it can sort in a minute. We approached this by letting our system sort different amounts of data, taking the total time and calculating how much of the data was sorted on average per minute. Due to the responsiveness of the DAS-5 limiting our input data size, the maximum input data used was 10.000.000 values, which is roughly 1Gb of data. Graph 6 shows our results.



Graph 6

Our peak performance lies somewhere around 1.75Gb per minute when sorting on 6 to 8 nodes. However, when we did manage to get a response from the DAS-5 after letting it sort 100.000.000 values, which is roughly 10Gb of data, the data sorted per minute appeared to have increased up to 2.5Gb per minute. This suggests that Graph 6 does not accurately portray the performance of our algorithm, and that it in fact performs quite a bit better when scaled up more. Unfortunately it is the best we can do at the present moment.

The current winner on sortbenchmark.org sorts 55Tb per minute. Their implementation runs on a distributed system of 512 nodes, significantly more than ours, but nonetheless it is safe to say that their algorithm outperforms ours by a mile.

Discussion

Implementing our system on the DAS-5 proved a lot more challenging than expected. Getting the algorithm to run locally was an easy task compared to getting it to run on the DAS-5. A large portion of our time was therefore spent debugging a relatively simple program. Due to this fact we unfortunately weren't able to implement any additional features that we had in mind. However, now realizing how daunting the task of implementing a distributed application is, we are somewhat satisfied with what we managed to achieve.

We were not able to perform all the experiments that we wanted to. We believe this is either due to the DAS-5's 'unwillingness' to process the large amounts of data we wanted it to, or due to some mistake in our implementation that we have yet to uncover. Unfortunately, this left us with results that are not quite satisfying and difficult to make claims about.

That our results are not able to match with the sorting implementations on sortbenchmark.org is no surprise. These implementations have been tweaked to perfectly match the hardware they are running on. Our implementation is able to run on heterogeneous hardware, and still deliver some kind of performance enhancement.

There are a few key aspects of our design that could have been improved upon. The gathering point is a global synchronization moment. This means that all nodes are waiting to receive the final data from the other nodes. One slow node could leave the other nodes idly waiting, which is obviously a huge waste of resources and precious computing time. This could be improved by alternating the gather and sort phase. This ensures that while nodes are not receiving data, they are sorting the current data. We expect that this would lead to a huge increase in performance. There would still be some synchronization, because eventually all nodes will have to wait for the final slow node, but there would be an improvement in terms of resource utilization.

Our implementation tries to optimize the packet size on the network speed, to send packets with the most optimal size. This however could lead to an unfair distribution of data in phase 1 and 2. This could reduce performance of the whole system as explained by the global sync point. More performance gain could be reached once we are able to make a perfect tradeoff between packet size and data distribution over nodes.

Furthermore our implementation was written in Python. It is no secret that this language is significantly slower than lower level alternatives. Further significant performance improvements could be made by porting this project to C. The reason we chose Python over C in the first place is because there is a development time vs execution time tradeoff. With python it is possible to do a lot more operations with a lot less code, so although it executes slower, it's faster to get to functional code. On top of that, we were already familiar with Python and had no real experience coding in, for example, C.

In the future we would like to run our algorithm on more nodes with larger input data to properly test how well our implementation scales.

Conclusion

Implementing a distributed application is difficult, even when all you are trying to do is sort some data. Instead of getting a single machine to do what you want, you are now trying to communicate between multiple machines in an attempt to coordinate them. The communication results in significant overhead that really throws a wrench in the application's performance.

We experienced all this and more during this project. However, we managed to implement a distributed version of bucket sort that actually works. Our experiments suggest that running our algorithm on multiple machines provides a performance gain in comparison to sorting on a single machine, which is the point of a distributed data processing system. The results also seem to suggest that our application scales, as it appears to improve in performance as more input data is distributed over more nodes. Unfortunately our experiments fell short of what we would have liked to achieve.

There are still various optimizations that could be implemented, like removing the global sync point or porting to a different language.
All in all, this assignment was a valuable introduction into the world of distributed computing and we will never again take it lightly.

Appendix A

Time sheet

Type	Time
Total time	116.5 hours
Think time	9 hours
Dev time	75 hours
Xp time	13 hours
Analysis time	3.5 hours
Write time	12 hours
Wasted time	4 hours