VRIJE UNIVERSITEIT AMSTERDAM

Design and Implement a Distributed Sorting Application

Hongzhong Yu, Junyan Li, Berk Aydaşgil
{h2.yu, junyan.li, b.aydasgil}@student.vu.nl

December 20, 2019

Abstract

As we learned in the distributed system class, there is rapid growth in demand for distributed ecosystem in business, education, and online gaming areas. It is the golden age of distributed ecosystems. In distributed applications, performance is one of the many significant features that software engineers might care about. So in the lab project of this course, our group chose to do the design and implementation of a distributed sorting application, which is a straight forward but interesting topic that can let us experience distributed system issues and realize distributed applications in practice.

Previously, we were quite familiar with the sorting algorithms, but unfamiliar with the distributed techniques. So we first designed the process of the distributed sorting. And then implemented it being scalability conscious. Conducted experiments of it on both DAS-5 and cloud services. To realize the performance of it, we also tested some sorting benchmarks on our application and compared it to some historical winners.

This document begins with a case definition and our prior assumptions before starting the implementation. After that, chapter 2 introduces the detailed design and implementation part of our lab project. We also provide one design alternative for small number of clusters and conclude the key features in our project. In the following chapter 3, we run our application on different kinds of the environments with different sizes of input data. And we also test the sorting benchmarks which are widely recognized. Finally, in chapter 4, we draw a conclusion to our lab project and discuss some future improvements.

Key words: Distributed System, Performance, Sorting Algorithm, Scalability, Cloud Services, Sorting Benchmark

CONTENTS

1 Introduction

Sorting operation is one of the most commonly used operations in data processing pipelines. There are many varieties of algorithms with different advantages and use cases. Some of them are, insertion sort, merge sort, quick sort, bubble sort. Other than the algorithm of choice, there are different configurations of the environment that the sorting takes place in. Data to be sorted can or can not fit into the main memory, and also in some cases, the data can not fit into a single machine altogether. When the data does not fit into the main memory, the sorting process is generally sought to be externalized. This has two main benefits. Firstly parallelized operation will result in speedups, and secondly, it will make sorting of data to be possible if it does not fit into a single machine. When the sorting is to be done in a distributed fashion, distributed aspect of it comes with many of its challenges. When dealing with distributed systems, some arising challenges are stragglers, fault tolerance, and bottlenecks(communication).

In our assignment of implementing a distributed sorting application, we abided to the details that are outlined on the website `www.sortbencmark.org`. The website includes information on top-performing systems, a variety of benchmarks, rules, and tools to qualify for benchmarking.

RULES Under the common rules section of the website, there are nine rules stated and described. We started the project initially focusing on two and later on increased the span. These were:

- Data must be read and written from the secondary storage using files.
- Evaluation time includes the launching of the sort program.

DATA GENERATION AND VALIDATION Input data for the sorting process was instructed to be generated with `gensort` record generator. After the sorting is done, the output is to be validated with `valsort`.

BENCHMARKS AND CATEGORIES There are many benchmarks that are equipped with different metrics. Some examples are minute sort, penny sort, joule sort. While minute sort measures sorting that takes place under a minute, penny sort measures sorting that has under penny's worth of system cost. Each and every one of the benchmarks have two categories to compete in. These are Indy and Daytona. Indy sort assumes key, and record sizes are fixed. And applications running under this category takes advantage of this assumption. Daytona sort is expected to be capable of sorting other records and key types, and while doing so, expected not to lose significant performance.

FAMILIAR TOOLS AND ASSUMPTIONS One of the tools that we were familiar with before starting the implementation was the C++ STL library. We chose to use this alongside with Open-MPI for its familiarity and also high-performance expectations compared to other alternatives. A sorting algorithm that came to mind was the k-way merge sort. We had prior knowledge of its advantages and expected it to perform well in our case. Since all of these preferences can be configured and used with DAS5, we decided to start the implementation with this feature set. Aligned with the rule from `www.sortbenchmark.org`, “the hardware used should be commercially available”, and to further assess performance variables of our implementation, we additionally planned to operate on the infrastructure of cloud providers. The cloud providers of choice were AWS and Vultr.

STRUCTURE OF THE ARTICLE In Design and the Implementation section of the article, we started off by describing the environment that we were working on, and the tools that we used. Following this, the Design of the implementation was described under two parts. And finally, the design alternatives were investigated, and our main focus, the key features were explained. Our evaluation of the implementation and the experiments were discussed under the Experimental Evaluation section. Here, the configuration of the environments, the achieved scalability with our implementation, cloud cost analysis, and results from a variety of benchmarks were shared. And in the final section, we concluded by summarizing our progress, sharing conceived future improvements to the implementation, and reporting our process.

2 Design and Implementation

In this part, we firstly introduce the detail of our design, including all the algorithms as well as the tools and platforms we used in our lab project. We continue with the key features in our design, as well as some other details in our implementation.

2.1 Environment and Tools

1. Development environment: All the code in our work is written on das-5 with the account provided by the tutor at the beginning of the course. We only used one of the accounts of das-5, which has 40GB disk storage. We'll mention the detailed information of the configuration in the evaluation part 3. And the detailed information can be found on the website.[2] We used this to store the data generated in our project.
2. Running environment: Same as the development environment, the running environment, and some debugging parts are also on das-5. After we've finished all the development, we also tested our application on two cloud service platforms, which are Amazon Web Services and Vultr. The detailed information can also be found in chapter 3.
3. Data generation and validation tools: The data generation tools and data validation tools are all existing tools for sorting the benchmark website. [11] We downloaded the Linux version and used the parameters *-a -size* to use the data generators. And use the validator to check whether the data is sorted in the right order.
4. Coding language and library: In our project, we chose C++ as our coding language. Since the sorting algorithm we want to use for our local sort, and the heap(priority queue) are all included in the C++ standard library. So we can use the *sort* and container *priority_queue* by simply included them. It saves our time in debugging some algorithm problems so that we are able to focus more on our distributed system part. The library we chose to use is OpenMPI. We learned some detailed ideas of OpenMPI in this course and noticed that it's super suitable for our work. And we also know that OpenMPI can run fast, and the sorting algorithms always care about the performance. So we chose to use it.

2.2 Design of Distributed Sorting

In this part, we are going to introduce the detailed design of our distributed sorting. To make things less confusing, we are going to introduce our design into two steps. In the first step, the sorting data will be generated by the master and then delivered to the slave nodes. We called it to deliver the data. And in the second step, the sorted data will be collected and merged by both the slave nodes and the master node and finally got the correct sorted order. We call it to merge the result.

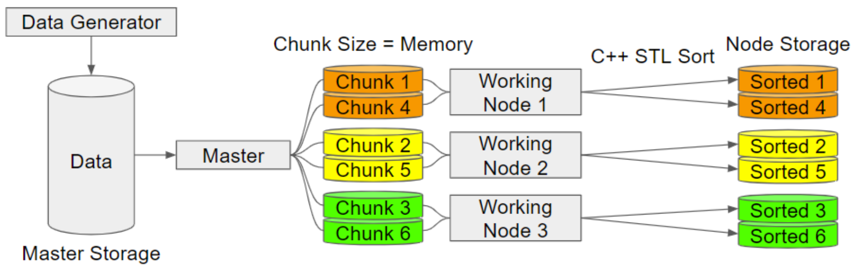


Figure 2.1: First step: deliver the data

2.2.1 First Step: Deliver the Data

As we've mentioned before, the first step of our application can be seen in Figure 2.1. The detailed process is as follow:

1. Firstly, we used the data generator on sorting the benchmark website to generate the data that needed to sort. All the data is stored in the Master node's storage.
2. After the data generation, the master node reads the data from its hard disk storage. Whenever the data reached the master's memory limit, which is set as the size of the chunk, the master will transmit this chunk of data to one of the slave nodes. And this part is done round-robin so that the following part can have a better parallel.
3. Finally, in the first step, when the slave node(or working node) receives the chunk of data from the master node. They use C++ STL sort to sort the chunk of data locally. And then they store the sorted data in their own storage.

We've finished the first step from data generation to storing all the data in size of chunk and store each sorted data in the storage of slave nodes. What we are going to do next is to merge all the data together to get the result.

2.2.2 Second Step: Merge the Result

Continue to the first step mentioned above, in this second step, and we are going to merge all the sorted data to get the final result. The process is in Figure 2.2. In this process, all the data should be merged twice. For the first time, the data on each slave node should be merged together by itself. And after that, the data from all of the slave nodes should be merged by the master node. This algorithm is called *K-way merge sort*. It means we have to get the smallest element among k different sorted set. To achieve this, we can simply use the heap(priority queue) to do this merge.

1. On the one hand, we use the *k-way merge sort* to merge all the sorted data chunk on the slave nodes. And every time, the slave node prepares the smallest K elements to form a package and send it to master.
2. On the other hand, the master maintains a heap(priority queue) that contains all the data received in packages from the slave nodes, so that the size of packages can be rep-

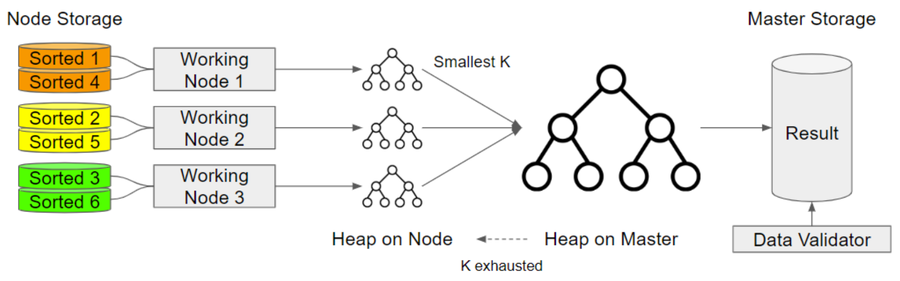


Figure 2.2: Second step: merge the result

resented as k = memory / number of slave nodes. These two processes can be running at the same time. When the master is merging, all the slave nodes are also merging their data. And whenever the master exhausted the k elements from a slave node, it will ask for k more from the same slave node.

3. Finally, the master used the heap(priority queue) to pop out the elements in the ascending order, and we ran the data validator to check whether the sorted result is in the right order.

In this way, we finished the merge of all the sorted data. And we finally get the result of our sorting algorithms.

2.3 Design Alternative

In this part, we provide another design alternative in a more simple way. This is the design we've achieved as the demo at first. But soon we found the bottleneck different as we've assumed before. So we change the structure. And we compare the differences in parallel between those two different designs. The design alternative has the process as follow:

1. Similar to our design, the design firstly generate the data with the data generator and store the data in master's storage.
2. And then the master node read the data from the storage. Whenever the size reaches the size of memory, the master node sends the data chunk to the slave nodes in a round-robin way.
3. The slave node only sort the data locally and then return the result to the master node. So that in the previous step, the master node has to receive the result first and then send data to that node.
4. When the master node receives the sorted chunk from the slave node, it saves that in its storage.
5. Finally, the master merges all the received sorted data locally and got the result.

And we can compare these two different designs. The differences between them are their performance, which is a result of the different parts in parallel.
For our design, we parallel the CPU running time with the disk IO time, which leads to a great

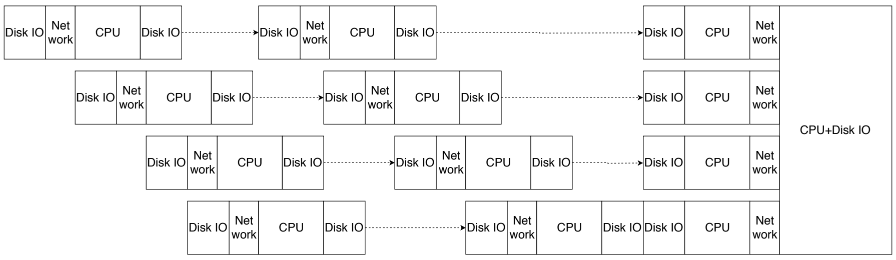


Figure 2.3: Process of Our Design

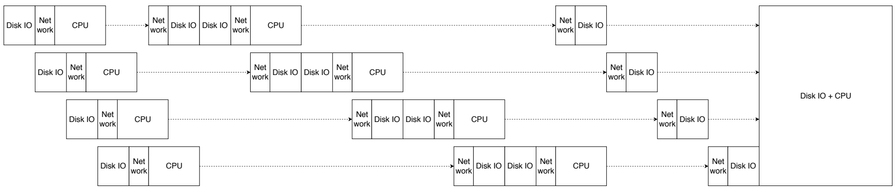


Figure 2.4: Process of Design Alternative

increase in performance. And if the application is running with more slave nodes, we can get a higher speedup.
As for the alternative design, it made the assumption that the network is slow in the process. So that this design parallels the network part. However, in all the platforms we've tested. The network is much faster than CPU and disk IO. But we didn't abandon this design; it runs faster than our design when there are only a few slave nodes. A probable reason is it actually sorted one less time than our design. So in our final application, if the slave nodes number is small, we'll use this structure to achieve higher performance. We dynamically choose which one to use.

2.4 Key Features

It's a very simple version of the distributed application. But we still tried to satisfy the following three key features during our implementation.

2.4.1 Strong Scalability

The first and most important feature of our application is strong scalability. As the master divides all the data into chunks and delivers them to each slave node round-robin. If we have more slave nodes, every node will need to sort fewer data. And the performance will increase. That means our application satisfies strong scalability. And in our evaluation part, we'll talk about some experimental results of this part.

2.4.2 High Performance

High performance is another feature we are really interested in. Our design process follows the following steps. Firstly, we design our application casually. Secondly, we implement a straightforward demo of our design. After that, we find the bottleneck and think about the way to get rid of it. Finally, we design again and iterate all the steps.
And we used two key algorithms and one library in our work:

1. C++ STL sort: It's one of the best implementation of sorting algorithm on a single machine. It's a combination of quick sort, heap sort, and insertion sort. It applies different sorting algorithms for different conditions and gets much better performance than using a single sorting algorithm.[13]
2. K-way merges sort: It's another sorting algorithm we used in our work. It looks like a combination of heap sort and merges sort. And it's widely believed that it's the best way to merge a number of sets of elements. [8, 9]
3. Open MPI: The library we use is Open MPI. We chose it at first because the tutor only mentioned this library. However, we later know that the benefit of Open MPI. It's considered to be one of the best performance libraries for the distributed system. [5]

So, in conclusion, every part of our work is using the best performing method. And our application should be relatively high performance among those who have the same hardware devices.

2.4.3 Fault Tolerance

We tried to implement two-part of fault tolerance in our application:

1. Hash Code: To avoid missing byte or transmission errors, we choose to send an additional hash code for every package of data we send between the nodes in our application. Whenever the code is not the same, the data will be re-sent again until the hash code matches.
2. Reschedule: Whenever one of the slave nodes is down, the application will be able to detect it and reschedule it's task equally to other nodes. (In our code, it's hard to achieve, and we are still fixing the bugs for this.)

2.4.4 Daytona Sort

Daytona sort means the sorting application is capable for sorting data with any length of keys.[7] It's a limitation to counting sort, bucket sort, as well as some low-level techniques. In our application, we get rid of those methods so that we easily achieved the requirement of Daytona sort.

2.4.5 Dynamic Structure

As a part of high performance, we used two different structures in our design and dynamically chose one of them based on node number and platform.

2.4.6 Cross Platform

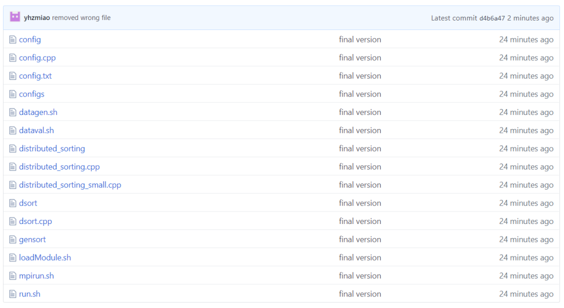
Our application is written in C++ with the standard library and Open MPI. And we also provide configuration tools for it to make it easy running and testing. So our application is quite easy to run across different platforms. It provides convenience for our experimental evaluation part.

2.5 Implementation Details

In this part, we are introducing some details in the implementation part of our work. We've made our source code public. It can be found on Github. See:

`https://github.com/yhzmiao/DistributedSorting`

1. *distributed_sorting.cpp*: This is the implementation part of our design. It can finish all the running part in our work.
2. *distributed_sorting_small.cpp*: This is the implementation part of the design alternative. It can also finish the running part in the work. We will choose to use it based on the platform and slave nodes we have.
3. *config.txt* and *config.cpp*: We can input the setting of file road, data to sort and some other things in config.txt. The config.cpp will generate all the command into *.sh* file and the code of sorting application in to *dsort.cpp*.
4. *dsort.cpp datagen.sh dataval.sh mpirun.sh*: These files are generated by config.cpp. They are the code and commands needed to run.
5. *gensort* and *valsort*: Tool provided by sorting benchmark.
6. *run.sh* It will run all the process with echoing some information for the steps.



yhzmiao removed wrong file		Latest commit d4b6a47 2 minutes ago
config	final version	24 minutes ago
config.cpp	final version	24 minutes ago
config.txt	final version	24 minutes ago
configs	final version	24 minutes ago
datagen.sh	final version	24 minutes ago
dataval.sh	final version	24 minutes ago
distributed_sorting	final version	24 minutes ago
distributed_sorting.cpp	final version	24 minutes ago
distributed_sorting_small.cpp	final version	24 minutes ago
dsort	final version	24 minutes ago
dsort.cpp	final version	24 minutes ago
gensort	final version	24 minutes ago
loadModule.sh	final version	24 minutes ago
mpirun.sh	final version	24 minutes ago
run.sh	final version	24 minutes ago

Figure 2.5: Files of our code

3 EXPERIMENTAL EVALUATION

3.1 EXPERIMENTAL SETUP

The experiments were run on DAS-5 and two Cloud IaaS services, Vultr and Amazon EC2. We ran one worker on one node to avoid interference on the speed of the I/O operation.

DAS-5 is a six-cluster wide-area distributed system running in multiple universities and organizations in Netherlands[2]. The system is designed to provide a common computational infrastructure. Users can submit workloads to DAS-5 compute nodes with the SLURM batch queueing system. The hardware specs of compute nodes in DAS-5 is shown in table 3.1.

Table 3.1: Hardware Enviornment of DAS-5

CPU	2x Intel Xeon E5-2630 v3 @ 2.4 GHz, 32 Threads		
RAM	64 GB	**Scratch Storage**	40 GB
Network	1 Gbps	**Local Storage**	7 TB

The IO speed in Local Storage is only half of the scratch. Because of the space limit in Scratch Storage[2], we could only perform up to 10GB data from scratch. The tests with more than 10GB input data were performed in Local Storage.

Vultr is a low budget global SSD cloud provider. Amazon EC2 is a web service that provides secure, and resizable compute capacity in the cloud[1]. The plans we used for the experiment are shown in table 3.2.

Table 3.2: Hardware Environment of Clouds

	Master (Vultr)	Worker (Vultr)	Master (EC2 t3a.xlarge)	Worker (EC2 x3a.large)
CPU	8x Virtual CPU 2.6 GHz	4x Virtual CPU 2.6 GHz	4x AMD EPYC 7571 2.2 GHz	2x AMD EPYC 7571 2.2 GHz
RAM	32 GB	8 GB	16 GB	8 GB
Storage	640 GB	160 GB	640 GB General Purpose SSD	160 GB General Purpose SSD
Network	1 Gbps	1 Gbps	5 Gbps	5 Gbps

The software environment in our cloud setups is summarized in table 3.3. The input data for the program was a file generated by gensort[12] based on needed records. The output was a file of sorted data, which was then validated with valsort.

Table 3.3: Software Environment

OS	Ubuntu 18.04.3 x64	**File System**	ext4
MPI	Open MPI 2.1.1	**File Sharing**	NFS
Compiler	g++ 7.4.0	**Gensort**	1.5

3.2 Strong Scalability

We tested strong scalability for our system on all three platforms. In this experiment, we offered fixed 10GB data as input. We increased the number of worker nodes from 1 to 10. The total execution time denoted here is the mean value. The experiment results are shown in Figure 3.1.

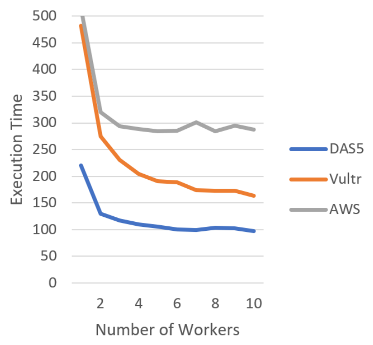


Figure 3.1: 10GB data, increasing workers

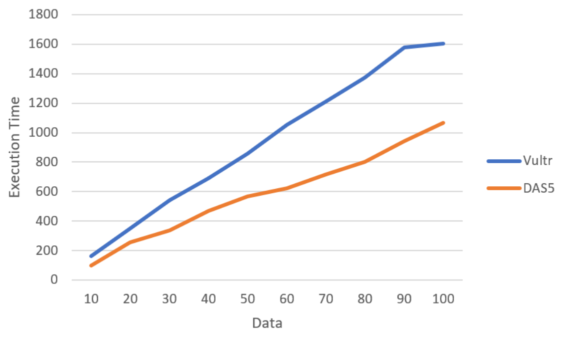


Figure 3.2: 10 nodes, increasing data

As we can see, when more workers are added, each worker sorts less data and results in less total execution time. This indicates strong scaling.

3.3 Increasing data for Fixed nodes

In this experiment, we provided an increasing amount of data from 10GB to 100GB for 10 worker nodes. The test results are shown in Figure 3.2, which indicates that our system was able to handle data that was larger than the DRAM size of a single machine.

3.4 Cloud Cost Analysis

Our experiments were performed on two cloud platforms, Vultr and Amazon EC2. Inspired by CloudSort benchmark[10], we discussed the cloud cost in our experiments on the GB scale.

In Table 3.4, we show the standard cost for the plans we used. We roughly calculated the per second cost for one master and 10 workers by dividing the hourly cost by 3600.

Table 3.4: Cloud Cost

	Vultr Master	Vultr Worker	AWS Master	AWS Worker
Container Cost	$0.238 / Hr	$0.06 / Hr	$0.1728 / Hr	$0.096 / Hr
Storage Cost	Included		$0.105 / Hr	$0.0033 / Hr
Per Second Cost (1 master + 10 workers)	$0.0002327777778		$0.000353	
Private Network Traffic	Free		$0.01 / GB Out	

We first calculated the average execution time of our test results. The mean time to sort 1GB data was 17.17s in Vultr and 28.75s in Amazon EC2. The per GB cost was then calculated with the following formula.

$$Cost/GB = Seconds/GB * Cost/Second$$

The resulting cost was $0.003996794 / GB for Vultr and $0.01014875 / GB for Amazon EC2. Amazon EC2 cost 2.5 times of Vultr.

We noticed that there was an extra cost in outbound private network traffic between containers in Amazon EC2. Sorting 1GB data consumed 2GB outbound traffic (1GB for transfer the data out from master, and 1GB for transfer sorted data out from workers). As a result, the actual cost of Amazon EC2 was $0.03014875 / GB, which is 7.5 times of Vultr. The reason for high cost and relatively low performance in Amazon EC2 was that EC2 set a hard limit on every part of the resources, such as CPU frequency and IO speed, while these were burstable in Vultr. Private network cost nothing more than the network device. It was offered for free on many cloud providers, while it was being charged at EC2 at a high price.

The cost metric was applicable for data less than 300GB. For data over 300GB, more storage on master and more workers were required, resulting in different cost metrics. We did not compare it with former Cloud Sort winners because they were using 100TB data, which was not do-able on our current experimental setup.

3.5 Sorting Benchmarks

In this section, we applied sort benchmarks with the experiment results. Since our experiment setups were based on a shared hardware environment, the execution time varied when the programs were executed with the same patterns at different times of the day. Hence we discussed the benchmarks with the best test results.

3.5.1 Minute Sort

Minute sort aims to sort as much as possible in one minute[4]. It indicates the number of gigabytes that can be sorted in a minute of elapsed time.

The best result in our record was 10.18 GB / minute, on DAS-5, with 10 workers. We did not obtain better results due to the limited number of worker nodes and the shared hardware environment. As a winning comparison, we defeat the past winners before 2000.

3.5.2 Penny Sort on Cloud

Penny sort aims to sort as much as possible for less than a penny's worth of system time[4]. Inspired by it, we considered the penny sort metric to be deployed on the cloud.

Based on the best result in Cloud Cost Analysis ($0.003996794 / GB in Vultr), our Penny Sort was calculated to be equal to 2.502 GB. This performance score is in closer range to the winner of 1999, which indicates that the cost for computation of a small scale is higher in the cloud than traditional hardware.

3.5.3 10GB,100GB Sort Time

Inspired by Datamation Sort, which indicates the amount of time to sort 100MB records[3], we measured the time to sort 10GB and 100GB records in our experiments.

The best sorting time was 58.95s for 10GB on DAS-5 with 10 workers, and 983.65s for 100GB on DAS-5 with 20 workers.

3.5.4 Joule sort

Joule sort indicates the amount of energy required to sort 10GB or 100GB records[14]. Since our experiment setup ran one worker on one node, we estimated the total power by summing up the TDP usage of a single CPU core under full load, a single RAM stick, and disk under read/write operation on DAS-5. The benchmark was not performed on cloud platforms because the details on their hardware configuration were unknown.

TDP of an Intel Xeon E5-2630 v3 is 85W[6]. We estimated the TDP of a single core under full load at 10.63W, disk under read/Write operation at 6.00W, and a DDR3 memory stick at 3.00W. The total TDP for a worker in DAS-5 was 19.63W. We considered the best sort time for 10GB (10^8 records) under the fewest workers in this experiment, which was 120.5s on three workers. The energy consumed was 1675.62J per node and 6702.48J in total. Thus, our metric was 14,919 records / Joule. With these results, we defeat the winner of CoolSort from the year 2007.

4 Summary

4.1 Conclusion

Now we can make a simple conclusion to all the work we did in this lab project. Firstly, we did some research about the sorting algorithms and read some books that introduce algorithms, which provided us some basic ideas of sorting algorithms, and we made sure about some detailed parts of the algorithms in our design. After that, we came up with two designs that have different structures. We implemented both of them and dynamically chose them in our project. To continue with the design part, we introduced the detailed implementation part of our work. And we also made our code open source. Finally, we did an experimental evaluation on three different platforms. We tested several benchmarks and compared it with some former winners.

4.2 Future Improvements

Here are three possible improvements that we aim to address in the future. Due to the time constraint and the limited resources, we encountered a lot of difficulties in design and implementation. So some parts of our work still need to be improved.

1. The first thing we plan to achieve in the future is further changing the structure of our lab project. We noticed that one of the structural bottlenecks in our design could be the merge sort on the master node. Although we parallelized the merge sort on the slave nodes, it still remains to be super slow compared to other parts in our work. So our plan is to change the *k-way merge sort* into a kind of bucket sort. So that we can get rid of the merge process. And it can further increase the performance of the sorting application.
2. There are still some bugs needed to be fixed in our fault tolerance part. It is hard to orchestrate the Open MPI under occurring fault since it does not provide the API for a timeout. Currently, we need to do that manually. And we also need to address the probable reschedule issues in every step of the application. It can still be improved.
3. Our experiments were performed on a small data scale because of insufficient hardware resources. We did not have desirable storage speeds in DAS-5, and we did not launch more cloud instances due to cost limitations. As a result, some sort benchmarks based on the 100TB scale did not fit our experiment results. In the future, we are looking forward to large scale experiments, if we can acquire access to more hardware resources.

4.3 Time We Spend

The time we spent on this lab project is listed as follows:

- Total time: The total time we spent on this project was about 132 hours. We have three members in our group, so it's about 44 hours per person. We did not need to observe the running experiments. So the effective time we spent was around **35-40 hours per person**.
- Time for Brainstorming and discussions: We spent 5-10 hours per person for brainstorming and coming up with ideas. So in total, it was about **20 hours**.
- Development time: We spent 15 + 10 + 10 hours in development. In total, it was about **35 hours**.
- Experiment time: We spent 5 + 10 + 5 hours in experiment. Some experiments ran relatively slow. And this declared time also includes time spent fixing bugs. In total, it was about **20 hours**.
- Analysis time: There were not a lot of work needed for analysis. And some of the analysis was done in the time we declared for brainstorming and discussions. We spent about 4 hours on discussion and analysis. In total, it was about **12 hours**.
- Writing time: We wrote our parts individually. And some of the ideas or figures were already used in our presentation. So including this, it took around 15 + 15 + 15 = **45 hours**, which also includes about 5-10 hours of preparation for the presentation.
- Wasted time: Occasionally, we lacked desired levels of communication, which made us slow down and brought up additional discussions that could be avoided. It was about 5 hours in total. And sometimes we had to put additional work to satisfy our requirements. For instance, some ugly figures or some code full of bugs. We wasted about 5 hours to redo the same thing. So in total, it took **10 hours**.

References

[1] Amazon. *EC2 Website*, (accessed December 18, 2019). `https://aws.amazon.com/ec2/`.

[2] V. U. Amsterdam. *Das-5 Website*, 2012 (accessed December 17, 2019). `https://www.cs.vu.nl/das5/accounts.shtml`.

[3] Anon. A measure of transaction processing power. 1985.

[4] Z. C. J. G. Chris Nyberg, Tom Barclay and D. Lomet. Alphasort: A cache-sensitive parallel external sort. 1995.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[6] Intel. *Intel Xeon E5-2600 v3 Product Family DataSheet*, 2014.

[7] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub. Tencent sort, 2017.

[8] D.-L. Lee and K. E. Batcher. A multiway merge sorting network. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):211–215, 1995.

[9] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[10] C. N. Mehul A. Shah, Amiato and N. Govindaraju. Cloudsort: A tco sort benchmark. 2014.

[11] C. Nyberg. *Sorting Benchmarks*, 2007 (accessed December 17, 2019). `http://sortbenchmark.org/`.

[12] C. Nyberg. *Gensort Data Generator*, 2011 (accessed December 18, 2019). `http://www.ordinal.com/gensort.html`.

[13] R. Robson. *Using the STL: the C++ standard template library*. Springer Science & Business Media, 2012.

[14] P. R. C. K. Suzanne Rivoire, Mehul A. Shah. Joulesort: A balanced energy-efficiency benchmark. 2007.