Dinesh Karamchandani A20407484

Sorting with Hadoop and Spark

a. Problem

This assignment extends from previous assignment (of external sort), as in this we are going to implement Hadoop and Spark versions. Here too, the main aim of this assignment is to sort files larger than the memory size, essentially trying to capitalize on sorting capabilities of the program by minimizing read and writes from/to the disk. We are again working with randomized data generated using Gensort suite (http://www.ordinal.com/gensort.html) and stored in '/input' folder on the Proton cluster. There are 3 such datasets of sizes 8GB, 20GB and 80GB. The program to be designed and implemented in Hadoop and Spark using Java as the programming language. We are also going to measure the performance of the above implementations and compare them against the performance results from the Linux "sort" command, from part PA2A of this assignment. The performance results comparison would be done for both strong scaling and weak scaling experiments as would be seen in the Performance section in this report.

b. Methodology

Since the main agenda was using Map Reduce for Hadoop and Spark, I have implemented them in the design for these 2 programs. Naturally, Java was used as the programming language.

Hadoop:

As Hadoop has the open source implementation of Map and Reduce, hence I created my own classes for Mapper and Reducer. In the mapper, the key and values are separated which in our case was the first 10 characters for key and the remaining 90 characters to be value in each line of the input file. I have used Identity Reducer as the reducer and have implemented 4 reducers and 4 mappers in the configuration. Also, I have used Snappy as the codec for compression as well as enabled intermediate file compression.

Spark:

In this part, since we are using the Java implementation of RDD and RDDPair to first store the input file into a RDD and then using mapToPair function we are getting the key value pair from each line of the input file. Finally, I am using reduceByKey, similar to, Hadoop implementation to get the sorted output.

Finally, the outputs from each of the above methodologies was also verified against TeraValidate program to validate the sort was done correctly or not. The output of this would be in part-r-00000 file which would have checksum if all the records are in order.

c. Runtime Environment settings

The runtime environment settings are as below:

i. *Operating System:*
Ubuntu 16.04.4 LTS, Release: 16.04, Codename: Xenial

ii. *Operating System kernel*
Linux proton 4.4.0-119-generic #143-Ubuntu SMP Mon Apr 2 16:08:24 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux

iii. *Java Compiler*
javac 1.8.0_162

iv. *Hadoop*
Hadoop 2.9.0

v. *Spark*
Spark version 2.3.0
Using Scala version 2.11.8, OpenJDK 64-Bit Server VM, 1.8.0_162

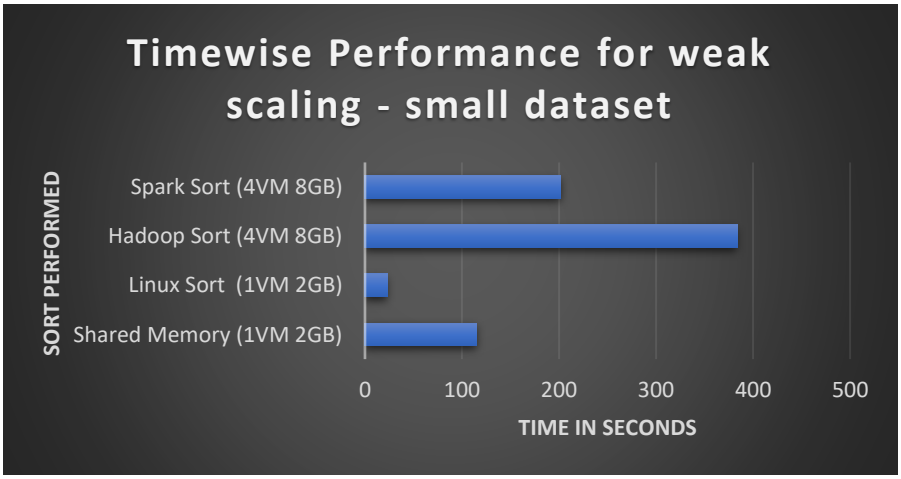
vi. *Make*
GNU Make 4.1

d. Performance Evaluation

The performance evaluation section showcases, the results of Hadoop sort and Spark sort programs as shown below.

1. Performance evaluation of sort (weak scaling – small dataset)

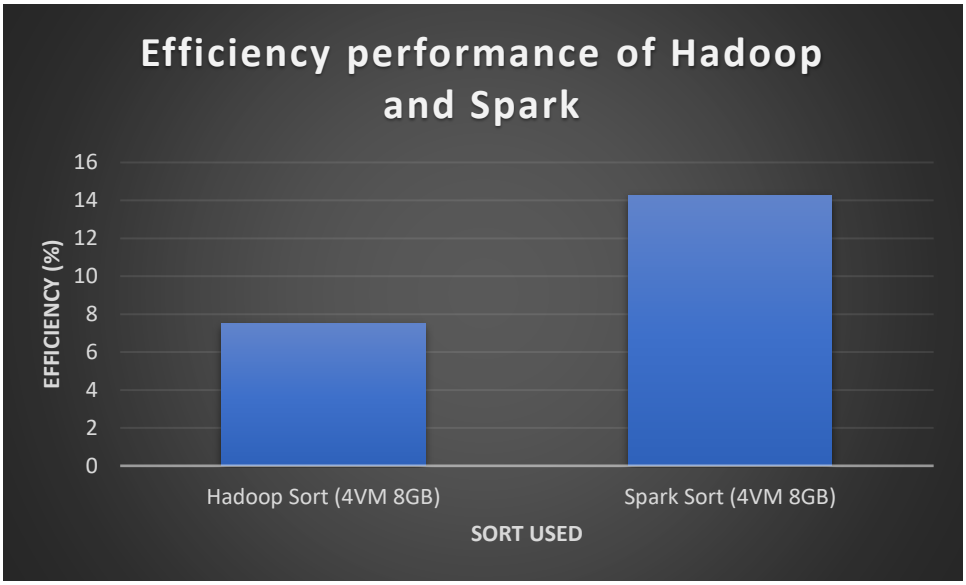
Experiment	Shared Memory (1VM 2GB)	Linux Sort (1VM 2GB)	Hadoop Sort (4VM 8GB)	Spark Sort (4VM 8GB)
Compute Time (sec)	115	23.04	383.849	201.632
Data Read (GB)	4	4	8.000487424	8
Data Write (GB)	4	4	8	8
I/O Throughput (MB/sec)	69.565	338.93	41.68432749	79.35248373
Speedup	NA	NA	0.299596977	0.570345977
Efficiency (%)	NA	NA	7.489924423	14.25864942



The graphs showcases the timewise performance achieved by the Hadoop and Spark implementations when compared with Shared memory and Linux sort.

As seen Hadoop sort takes the most time and has lower efficiency as seen in the table above when compared with Spark's efficiency which is better and almost 2X times Hadoop.

Below graph showcases the efficiency performance of Hadoop and Spark both for the above experiment



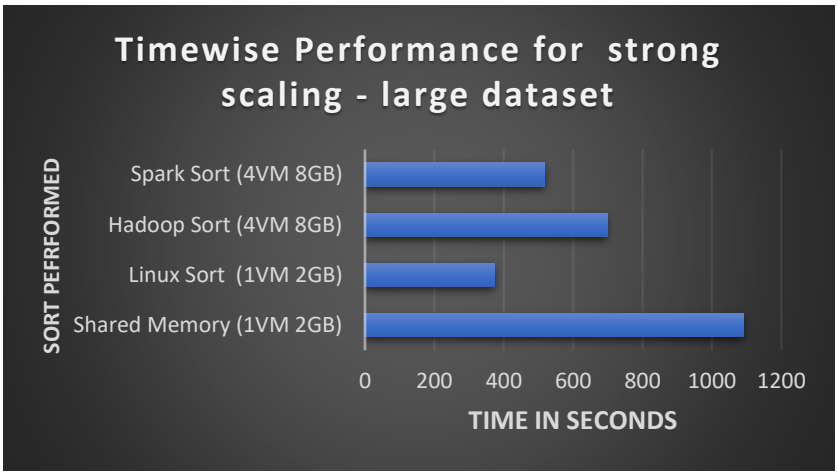
As seen in the graph, efficiency of Hadoop is lower than that of Spark and is almost 2 times lesser than Spark, this is probably due to the fact that since Spark performs in-memory all the data was sorted in its memory whereas for Hadoop it was by default sorting from disk.

2. Performance evaluation of sort (strong scaling – large dataset)

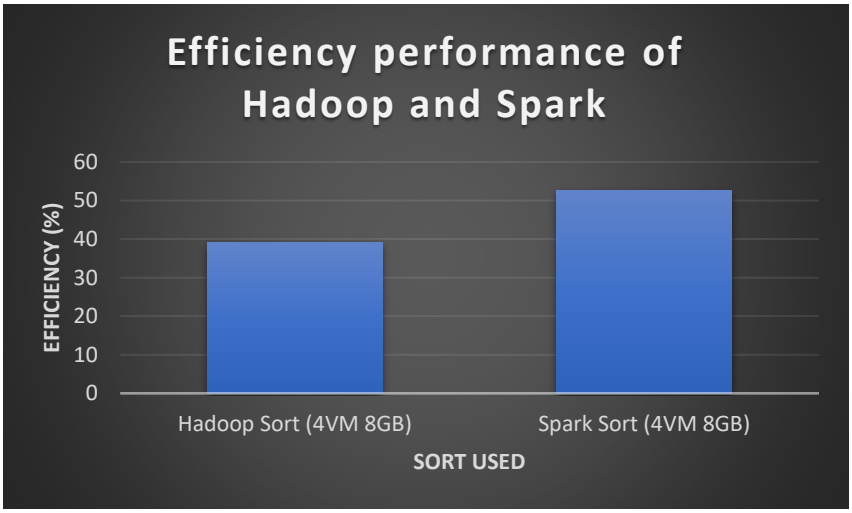
Experiment	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)	Hadoop Sort (4VM 20GB)	Spark Sort (4VM 20GB)
Compute Time (sec)	1092	374.121	698.718	518.825
Data Read (GB)	40	40	20.00121651	20
Data Write (GB)	40	40	20	20
I/O Throughput (MB/sec)	73.26	213.83	57.24944328	77.09728714
Speedup	NA	NA	1.562862271	2.104755939
Efficiency (%)	NA	NA	39.07155677	52.61889847

The graph showcases, the timewise performance achieved by the Hadoop and Spark implementations when compared with Shared memory and Linux sort

As seen below Shared Memory sort has taken the most time in this experiment as this is dealing with large dataset. Hadoop and Spark implementations both are better performing in this case.



Below graph showcases the efficiency performance of Hadoop and Spark both for the above experiment



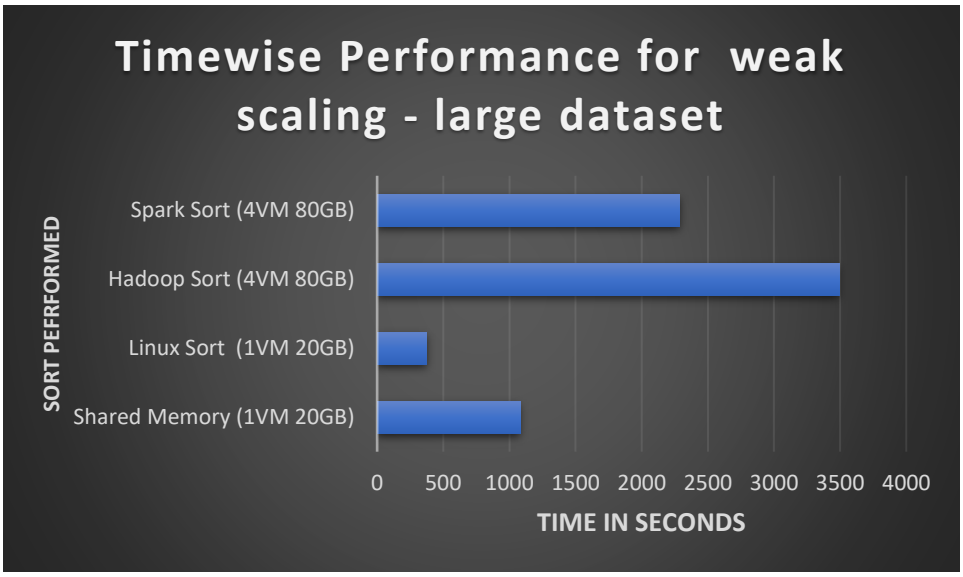
As seen in the graph, efficiency of Hadoop is lower than that of Spark and is almost 2 times lesser than Spark, this is probably due to the fact that since Spark performs in-memory all the data was sorted in its memory whereas for Hadoop it was by default sorting from disk.

3. Performance evaluation of sort (weak scaling – large dataset)

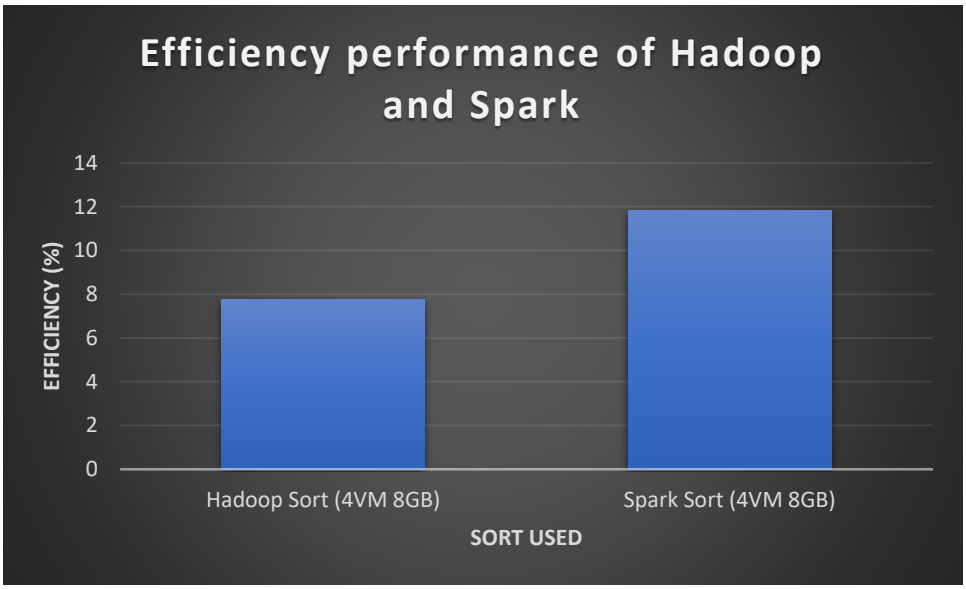
Experiment	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)	Hadoop Sort (4VM 80GB)	Spark Sort (4VM 80GB)
Compute Time (sec)	1083	374.121	3492.18	2287.484
Data Read (GB)	40	40	80.03567924	80
Data Write (GB)	40	40	80	80
I/O Throughput (MB/sec)	73.86888273	213.83	45.8268701	69.94584443
Speedup	NA	NA	0.310121471	0.473445934
Efficiency (%)	NA	NA	7.753036785	11.83614836

The graph showcases, the timewise performance achieved by the Hadoop and Spark implementations when compared with Shared memory and Linux sort

As seen below Hadoop sort has taken the most time in this experiment as this is dealing with large dataset. Hadoop and Spark implementations both are better performing in this case.



Below graph showcases the efficiency performance of Hadoop and Spark both for the above experiment



As seen in the graph, efficiency of Hadoop is lower than that of Spark and is almost 2 times lesser than Spark, this is probably due to the fact that since Spark performs in-memory all the data was sorted in its memory whereas for Hadoop it was by default sorting from disk.

Dinesh Karamchandani A20407484

Conclusions:

1. What conclusions can you draw?

Ans: From the results, I can say that there is a co-relation between the dataset size and the performance. For datasets lesser than memory size, Shared Memory performs the best whereas if we look at datasets having greater than memory size Hadoop and Spark implementations using MapReduce techniques are the better performers in this space.

Also, some of the key challenges involved in executing this sorting was the use the of multiple mappers and reducers of 4 each in this case along with compression of intermediate outputs and getting map tasks to take the maximum time to timeout. These are some of the reasons which are not yet fully explored and optimized by me, there is yet more ground that could have been covered if I could have implemented a way to reduce the final merge/reduce pass for Spark.

2. Which seems to be best at 1 node scale?

Ans:

- If dataset size to be sorted is lesser than memory size, then Shared Memory Sort is the best option to sort the data.
- Else if dataset size to be sorted is greater than memory's size, then Hadoop or Spark is the best option to sort the data. Also, if we are able to reduce the time taken for final merge or Reduce pass for then Spark is the best as Spark uses in-memory sorting which is always faster than Disk based External Sorting in Hadoop

3. How about 4 nodes?

Ans: For 4 nodes, I would say Hadoop or Spark both are the best options to sort the data. If we can reduce the time taken for the final merge phase or reduce phase for Spark, then Spark's in-memory sorting would provide better benefits and performance than Hadoop's disk based external sorting.

4. Can you predict which would be best at 100 node scale?

Ans: Although MapReduce has a tremendous capability, in Hadoop it uses external memory sort i.e. the approach of sorting is via the use of disk whereas Spark uses in-memory sort i.e. its approach to sort data in memory itself. As the speeds of memory are much higher than the speeds of SSD, hence I would say Spark is my best bet on working best on 100 node scale.

5. How about 1000 node scales?

Ans: If we only take in to consideration the performance factor of the sorting, then I would again recommend Spark as the better performer at 1000 node scale.

6. Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark.

2013, 1.42 TB/min **Hadoop** 102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) Thomas Graves Yahoo! Inc.	2014, 4.27 TB/min **Apache Spark** 100 TB in 1,406 seconds 207 Amazon EC2 i2.8xlarge nodes x (32 vCores - 2.5Ghz Intel Xeon E5-2670 v2, 244GB memory, 8x800 GB SSD) Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia Databricks

As seen from the table above the hardware requirements of both these winners is very different than the hardware requirements provided to us. Also, these winners have had great experience in using Hadoop and Spark that too on a competitive basis. So naturally they would have superior performance for their sorting algorithms.

If we only compare our results number-wise with the above even then the our performance is sub-bar as the winner can sort data in the range of 1TB/min whereas we are not even able to achieve 80 GB in under 30 mins. Also, another point to note is that Spark winner has better results than Hadoop which we can interpret as Spark being the better performer which is same as my inference above from my experiments.

7. what can you learn from the CloudSort benchmark, a report can be found at (http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf).
   The primary goal of the cloudsort benchmark is on the implementation of external sort. This benchmark is like a competition to get the least time to sort a large file typically 100TB or more along with the cost associated with doing the same. This benchmark also helps in finding the most cost-effective implementation of performing sort using various technologies in varying environments.