CS-553- PA2B
REPORT

Krishna Bharadwaj
A20398222

Contents

1 Introduction 3

1.1 Problem Statement 3
1.2 Implementation details 3
1.2.1 Hadoop 3
1.2.2 Spark 3

2 Performance Analysis 4

2.1 weak scaling – small dataset 4
2.2 strong scaling – large dataset 6
2.3 weak scaling – large dataset 8

Conclusion 10

1 Introduction

This document gives a detailed performance analysis of the terasort implementation on Hadoop and Spark

1.1 Problem Statement

Implement terasort on a cluster with Hadoop and spark. Compare the performance of these sort implementations against the shared memory sort implementation and the linux. Perform strong scaling and weak scaling experiments and analyze the results.

1.2 Implementation details

1.2.1 Hadoop

Implemented the Hadoop sort a cluster using the total order partitioner class provided by Hadoop. The algorithm is implemented as follows –

1. Perform a map only task by setting the partition class as the total order partitioner. An InputSampler samples keys across all input splits, and sorts them using the job's Sort Comparator. I have used the random sampler in order to sample the data
2. After the shuffle, each reducer has fetched a sorted partition of (key,value) pairs from each mapper. At this point, all keys in reducer 2 are alphabetically greater than all keys in reducer 1, which are greater than all keys in reducer 0. Each reducer merges their sorted partitions (using a sorted merge-queue) and writes their output to HDFS.
3. It can be observed that we are performing 2 reads and 2 writes in order to perform the sort

1.2.2 Spark

1. Fetch the keys from the lines, i.e. the first 10 characters
2. Use the inbuilt sortByKey function in spark to perform the sort
3. The output data is flattened by mapping to a flat map and written to disk

2 Performance Analysis

This section performs a detailed analysis of the experiments that were conducted and the results obtained.

2.1 weak scaling – small dataset

The table below summarizes the data obtained on performing weak scaling experiment on Hadoop and Spark. We increase the dataset while keeping resources constant.

	Shared Memory (1VM 2GB)	Linux Sort (1VM 2GB)	Hadoop Sort (4VM 8GB)	Spark Sort (4VM 8GB)
Computation Time (sec)	86.47	20	367.665	403
Data Read (GB)	4	4	16	8
Data Write (GB)	4	4	16	8
I/O Throughput (MB/sec)	92.51763617	400	87.03575266	39.70223325
Speedup	NA	NA	0.235186923	0.214565757
Efficiency	NA	NA	0.058796731	0.053641439

Table 1: weak scaling – small dataset

It is observed that shared memory has a lesser compute time than Spark and Hadoop. This is mainly because –

1. We were restricted to running Hadoop and spark only on 4 nodes
2. My Hadoop implementation ran 2 map jobs and 1 reduce job, in order to maintain the order of the sorted output file.
3. Shared memory was implemented on a 2GB dataset, whereas Hadoop and sort were implemented on an 8GB dataset. Even though we had 4 nodes to work with, at one point in the sort, there is a sequential bottleneck. Hence we see an increase in time

The chart below compares the compute time of each of the implementations. As expected, linux sort performs the best.

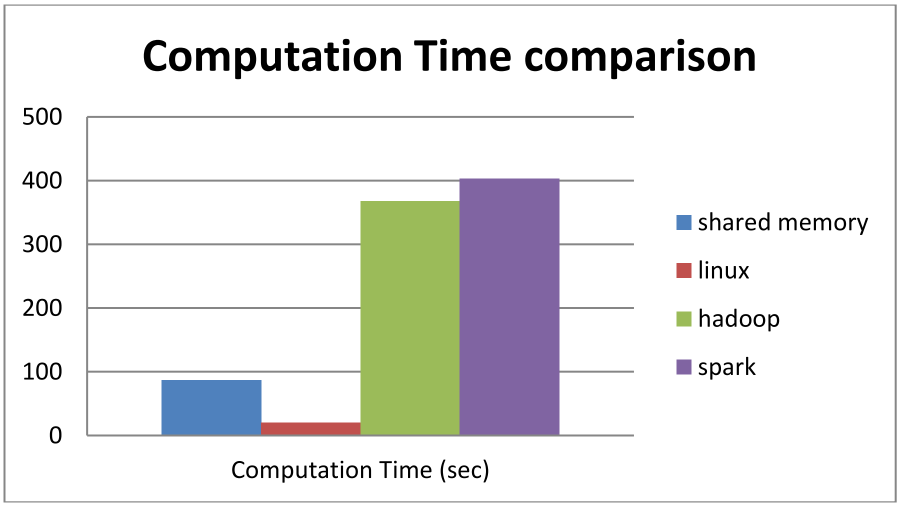

Chart 1 Computation time comparison

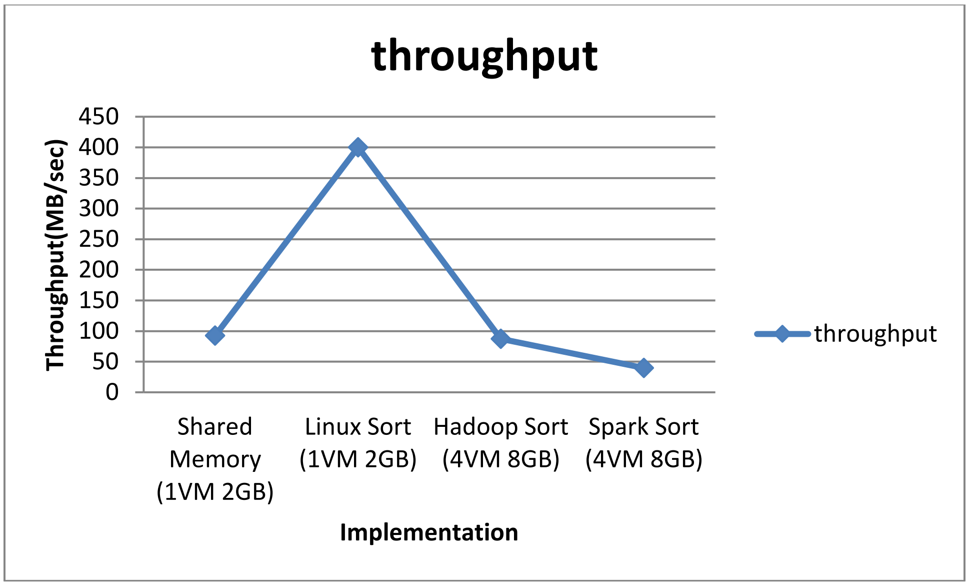

Chart 2 Throughput comparison

The above chart compares the throughput of each of the experiments. We see that Hadoop and shared memory sort have a similar throughput when it comes to weak scaling experiments on a small dataset.

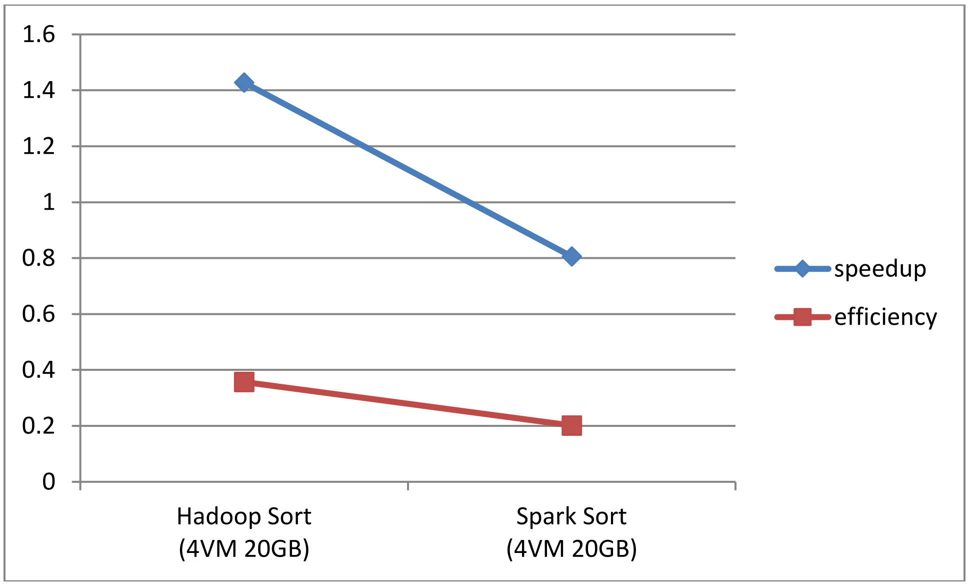

Chart 3 Hadoop Vs Spark

The chart above compares the speedup and efficiency values of both Hadoop and spark. Surprisingly, Hadoop is performing better than spark. This is mainly due to my implementation. I was not able to implement an efficient total order partition algorithm in Hadoop, thus utilizing multiple reducers.

Spark implementation was just the usage of sortByKey()

2.2 strong scaling – large dataset

The table below summarizes the data obtained on performing weak scaling experiment on Hadoop and Spark. We keep the dataset constant while changing the resources.

	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)	Hadoop Sort (4VM 20GB)	Spark Sort (4VM 20GB)
Computation Time (sec)	1069.808	419	749.456	1062
Data Read (GB)	40	40	40	40
Data Write (GB)	40	40	40	20
I/O Throughput (MB/sec)	74.77977357	190.9307876	106.7440917	30.12048193
Speedup	NA	NA	1.427446041	1.007352166
Efficiency	NA	NA	0.35686151	0.251838041

Table 2 String Scaling - Large Dataset

In this experiment, we see that Hadoop and spark perform better than shared memory sort. This is because –

1. Fixed data size
2. Hadoop and spark are running on multiple nodes, hence exploiting parallelism to run faster

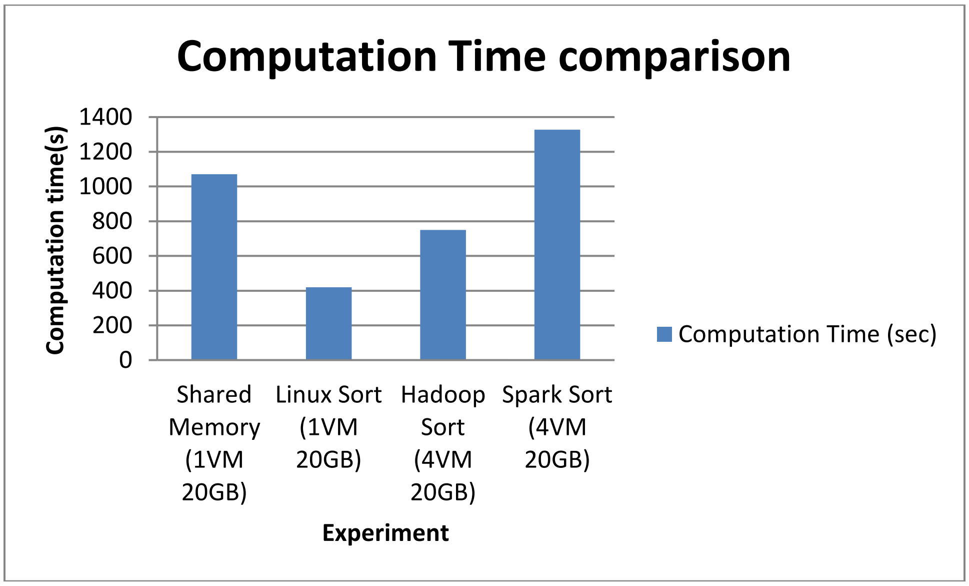

Chart 4 Compute time comparison

The chart above compares the compute time of each of the implementations. As expected, linux sort performs the best. However, for a fixed data size, Spark and Hadoop perform better than shared memory.

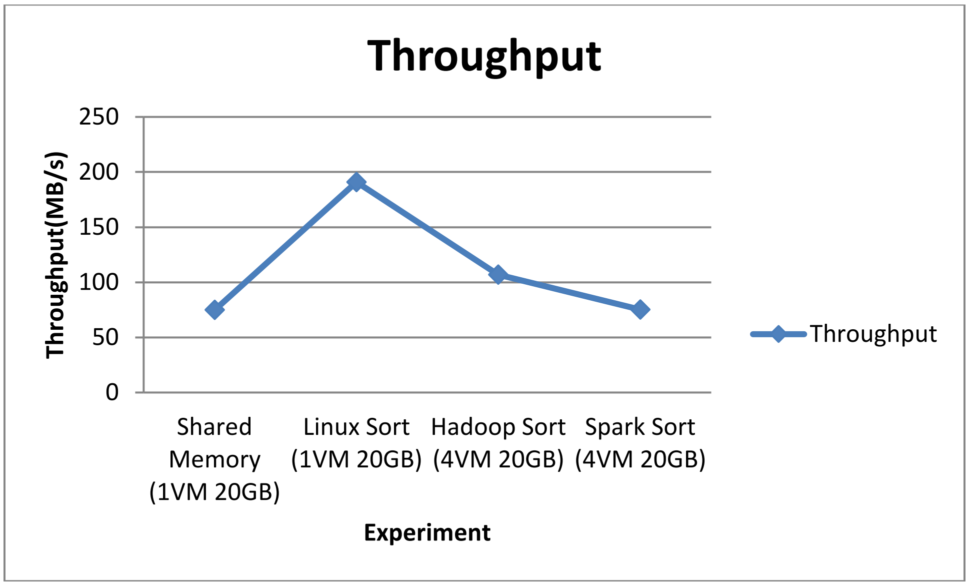

Chart 5 Throughput comparison

The above chart compares the IO Throughput. The throughputs of Hadoop and spark are better compared to the shared memory sort implementation.

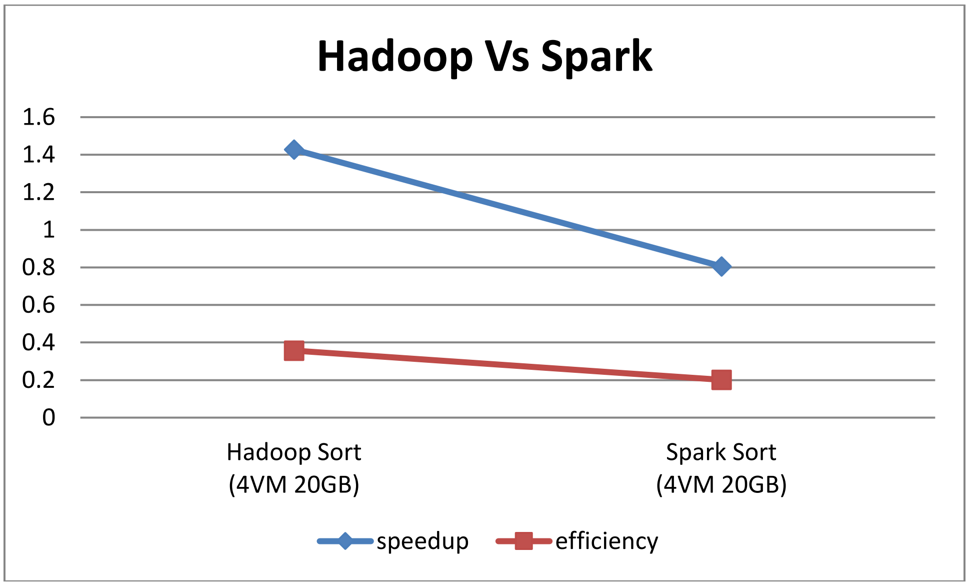

Chart 6 Hadoop VS Spark

The chart above compares the speedup and efficiency values of both Hadoop and spark. Surprisingly, Hadoop is performing better than spark. This is mainly due to my implementation. I was not able to implement an efficient total order partition algorithm in Hadoop, thus utilizing multiple reducers.

Spark implementation was just the usage of sortByKey()

2.3 weak scaling – large dataset

The table below summarizes the data obtained on performing weak scaling experiment on Hadoop and Spark. We increase the dataset while keeping resources constant. We select the larger datasets to check how Hadoop and spark scale with increasing datasets while compared to Shared memory sort

	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)	Hadoop Sort (4VM 80GB)	Spark Sort (4VM 80GB)
Computation Time (sec)	1069.808	419	3467.31	3017.07
Data Read (GB)	40	40	160	80
Data Write (GB)	40	40	160	80
I/O Throughput (MB/sec)	74.77977357	190.9307876	92.29056531	53.03158362
Speedup	NA	NA	0.308541203	0.354585078
Efficiency	NA	NA	0.077135301	0.088646269

Table 3 weak scaling - Large Dataset

In the weak scaling experiment with larger datasets, Hadoop and spark seem to perform better than the shared memory sort. This is because –

1. Less number of IO access per node as compared to the shared memory implementation
2. Increased parallelism has overcome the sequential bottleneck

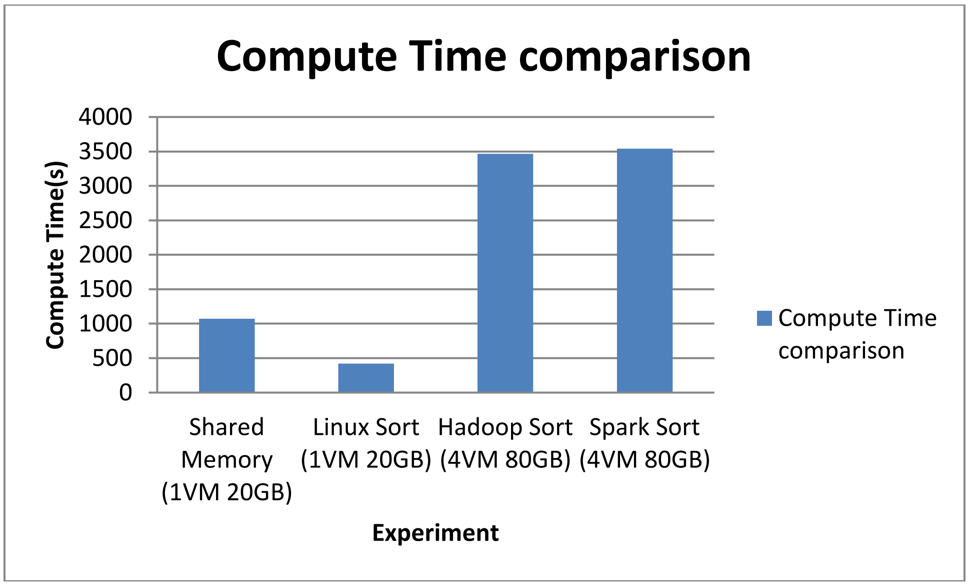

Chart 7 Compute time Comparison

The above chart compares the compute times for each of the implementations. It is evident that Hadoop and spark scale better than the shared memory sort,

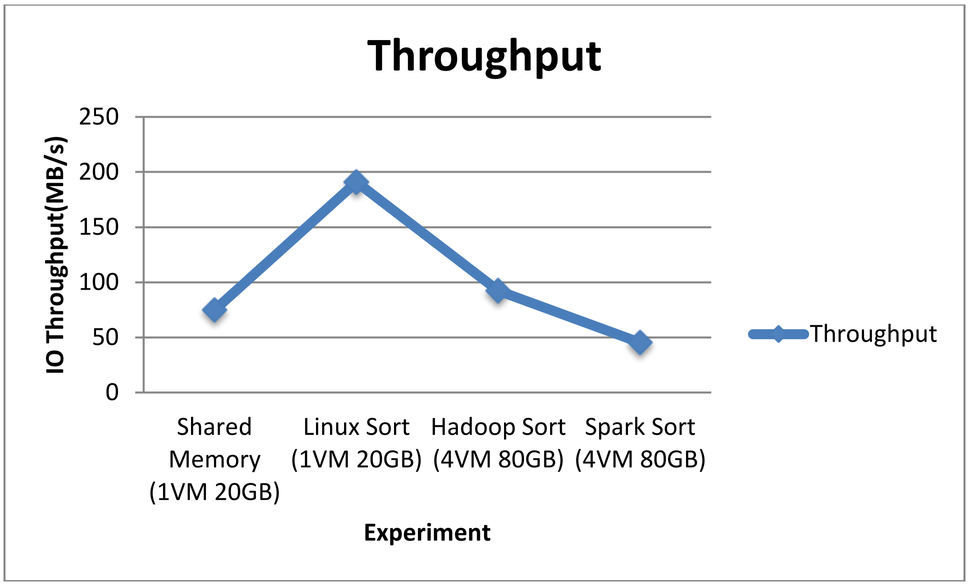

Chart 8 Throughput Comparison

As seen above, the throughput of Hadoop and spark are higher than shared memory sort.

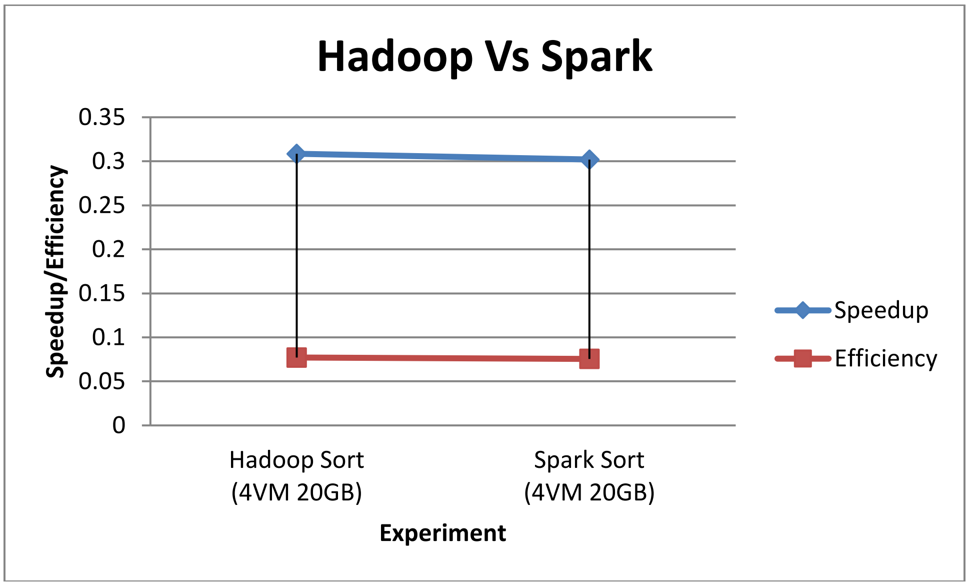

Chart 9 Hadoop Vs Spark

The above chart compares the efficiency and throughput values of Hadoop and spark implementations for the weak scaling experiments.
Surprisingly, Hadoop is performing better than spark. This is mainly due to my implementation. I was not able to implement an efficient total order partition algorithm in Hadoop, thus utilizing multiple reducers.

Spark implementation was just the usage of sortByKey(), without any tuning or optimization.

Conclusion

As part of the experiment, the following can be deduced –

1. Hadoop and spark have similar performance at 4 nodes
2. As the number of nodes increases, Spark's performance would be better than Hadoop since more nodes equals more memory. Since Spark performs all its tasks in memory, Spark's performance would increase