Sort on Hadoop/Spark - Design Document

In this assignment, we have to sort varied size of files on Amazon AWS using our own implementation, Hadoop and spark.

SharedMemorySort

This is my own implemented code in Java using quick and merge sort. I have kept the no. of files to be created as constant to perform sort on. I am increasing the no. of files for datasets more 10GB. I have used multithreading to perform quick sort in which equal no. of files will be divided among the threads.

Program Flow:

Main Class:

1. Calculate the chunk size by using input file length and no. of files.
2. Read the input file and split them into files with chunk size. Store the file names in a list.
3. Iterate the array to initialize each thread of another inner class "ReadSort" and start them.
4. Call the MergeSort function to merge all the files using merge sort implementation

ReadSort Class:

1. Read the chunk file into a list and sort them in the memory.
2. Write the sorted array into the same input file.

In the k-way merge, I am reading first line from all the files and writing the smallest one in to the final sorted file. Then I am reading the second line from the file which contains the smallest line. Repeating these steps one of the file will reach the end of file and at that time I will take all the remaining files data and perform sort and write collectively.

HadoopSort

In this code I have created only one class which is mapper. The mapper is to map each line of the file into key pair. Since I don't need any value part I have used null value. The reducer is not required in this case so I haven't created any reducer class explicitly. I am using an Identity reducer for default.

Program Flow:

Main Class:

1 Create file input and output objects with parameters using command line arguments.
2 Set the mapper class which I have created and reducer class to IdentityReducer
3 Set output key class as Text class and output value class as NullWritable class
4 Run Job

MapSort Class:

1 Collect Key as line and Value as Null and return

In Hadoop, I have used c3.large instance for single node and for 16 nodes, c3.2xlarge instance for master and c3.large instance for slaves with additional EBS volume of 50GB.

SparkSort:

For spark, I have implemented code in Java but since the Java output writes key value pair along with brackets in the file, I ran the commands directly on the spark shell.

Program Flow:

1. Create spark and Java configuration objects.
2. Read the input file name from command line argument and assign it to Java RDD object
3. Split the file by each line and map the line as a key and null as value in another object called as JavaPairRDD
4. Use the default sortByKey() method to sort the JavaPairRDD
5. Save the RDD in a text file

In spark, I have used c3.large instance for single node experiment and for 16 nodes I have used c3.4xlarge instance as master and c3.large instance as slaves with additional 50GB of EBS. Earlier I used 30GB of EBS for slaves but that resulted in low disk space and the code was stuck in second phase so I have to use 50GB of EBS.

Environment Setup:

Software:

- Java version 8.0
- Hadoop version 2.7.2
- Spark version 1.6.1
- Gensort version 1.5

Operating System: Ubuntu & Amazon Linux

System: Amazon AWS instances: c3.xlarge, c3.2xlarge, c3.4xlarge, d2.large, t2.micro

Steps:

1. Install the “intall.sh” script in t2.micro instance which is free of charge
2. Configure the below files of Hadoop:
   Core-site.xml, hdfs-site.com, mapred-site.xml, yarn-site.xml, Hadoop-env.sh except slaves (Only on Master).
3. Take an AMI image of the current configuration.
4. Use this image to launch Amazon on-spot instances for Hadoop and Spark.

Performance Evaluation

In this experiment, I have performed various experiment on c3.large instance of Amazon AWS to sort files of varied sizes using different implementations

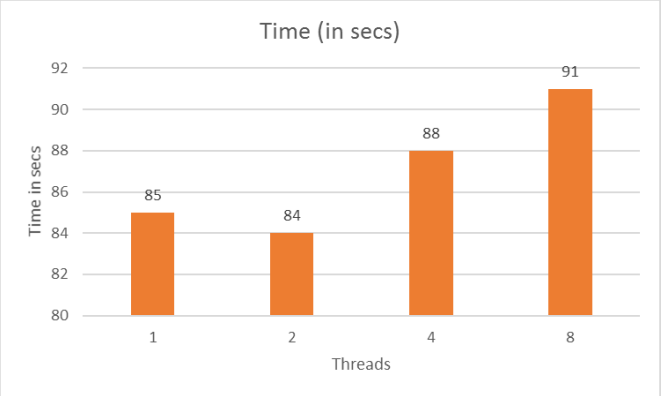
SharedMemorySort

File of 1GB with multithreading on single node of c3.large

Time:

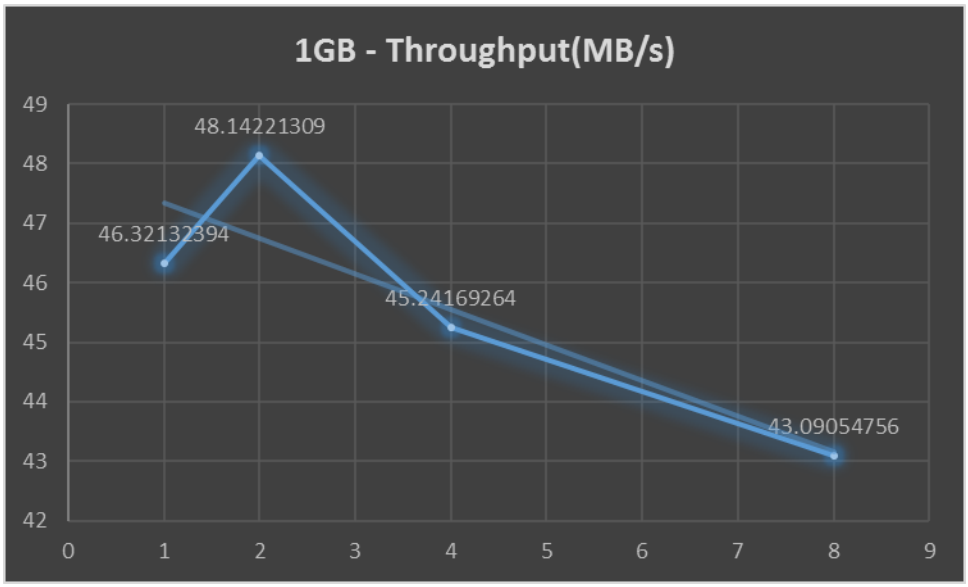
In this experiment, I ran my shared memory code on c3.large instance for 1GB with 1,2,4,8 threads. Since the dataset was very small, I didn't find any major fluctuations in the sorting algorithm. For all the threads, the time was almost the same which can be depicted from the below table.

Time (in secs)				
1GB	Thread 1	Thread 2	Thread 4	Thread 8
Gensort	14.76	14.76	14.76	14.76
Split	25	25	27	26
Quick	35	32	35	38
Merge	50	52	53	53
Valsort	16.52	14.06	13.92	13.99



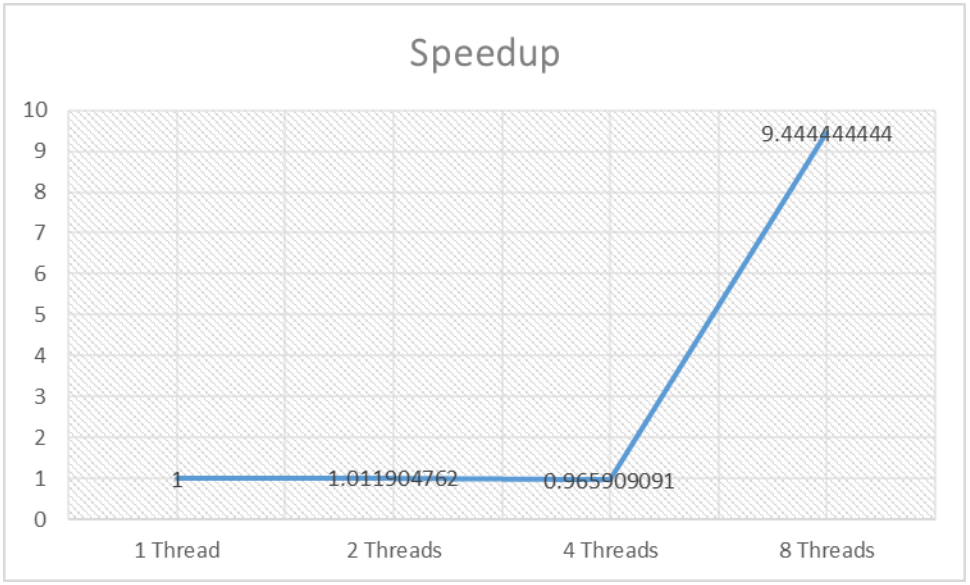
Throughput:

From the above explanation and table, I calculated the throughput achieved in each case. I found that for threads scaling from 1-8 the throughput range from 43 – 48 Mega Bytes per second (MBps).



Speedup

Below speedup is sublinear till 4 threads but it goes super linear for 8 threads.

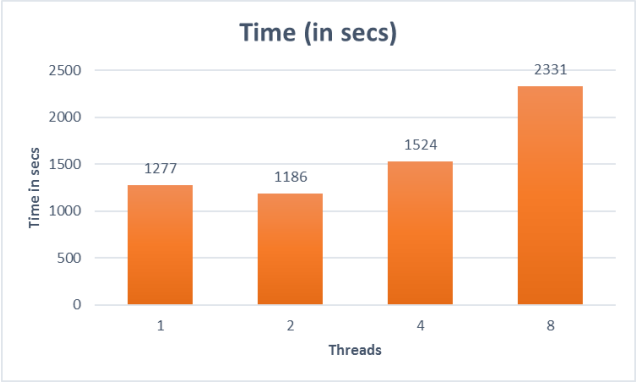


File of 10GB with multithreading on single node of c3.large

In this experiment, I increased my dataset to 10GB and ran my algorithm with 1-8 threads. Below is the table of time taken in each experiment for different set of tasks. For dataset creation and validation it takes around 2 minutes whereas for splitting it takes around less than 3 minutes.
Threading is only implemented in quick sort. Since there are only 2 cores in c3.large instance of amazon the code runs well for 2 and 4 threads but due to thread switching it takes a little longer time for 8 threads but overall the time is less as compared with single thread.
The k-way merge sort works only with single threaded but the performance of merge sort fails if there are large no. of files to merge. Hence, it can be seen that for 1, 2 and 4 threads the time taken was comparatively lower than time taken by 8 threads. In 8 threads I have increased the no. of files to match according to the main memory of the system which was only 3.75GB and JVM gets only a part of it.

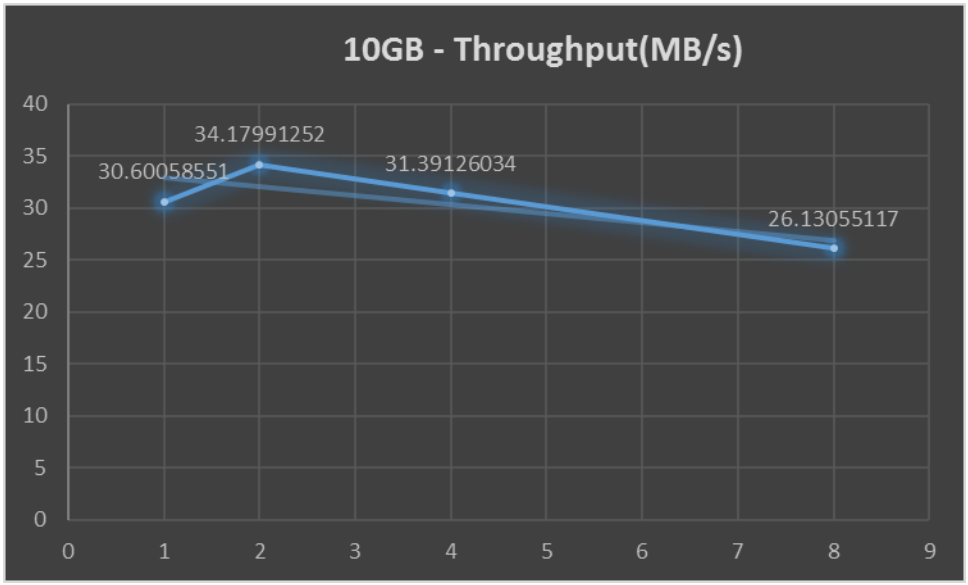
Time:

Time (in secs)				
10GB	Thread 1	Thread 2	Thread 4	Thread 8
Gensort	160	159	160	163
Split	277	282	291	280
Quick	540	449	419	453
Merge	737	737	1105	1878
Valsort	161	160	158	161



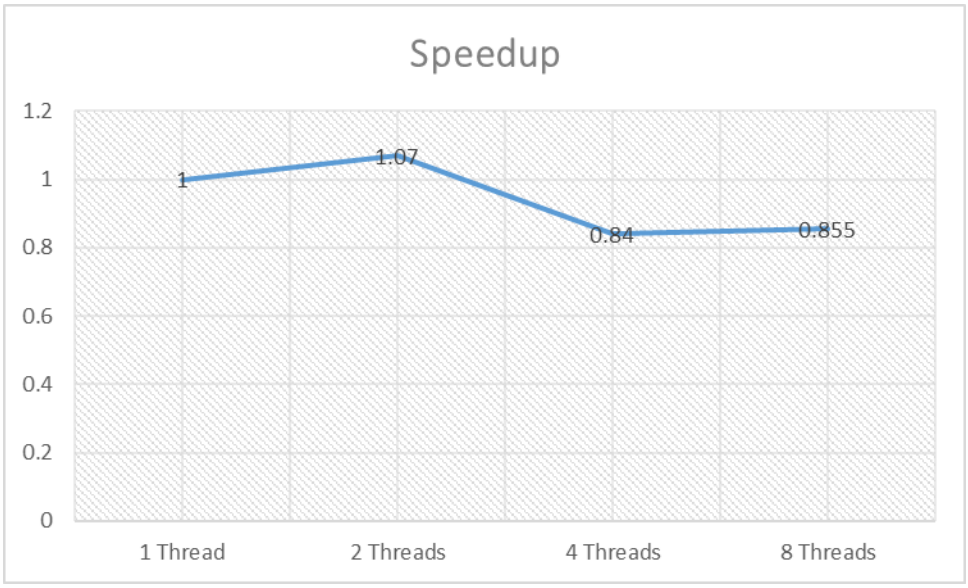
Throughput:

Below is the graph of throughput achieved with 10 GB datasets with multi-threaded system. As per c3.large system the program gives better output with 2 threads which is around 34 Mega Bytes per second (MBps).



Speedup:

Below speedup is sublinear for all the no. of threads.



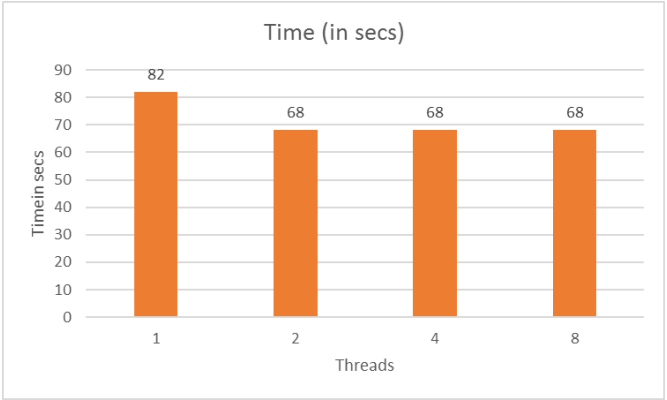
File of 1GB with multithreading on single node of d2.xlarge

This experiment was performed on d2.xlarge instance of Amazon AWS. It contains 4 CPUs with 30.5 GB of memory. Hence, I found my code took less amount of time to run compared to the above experiments.

Time:

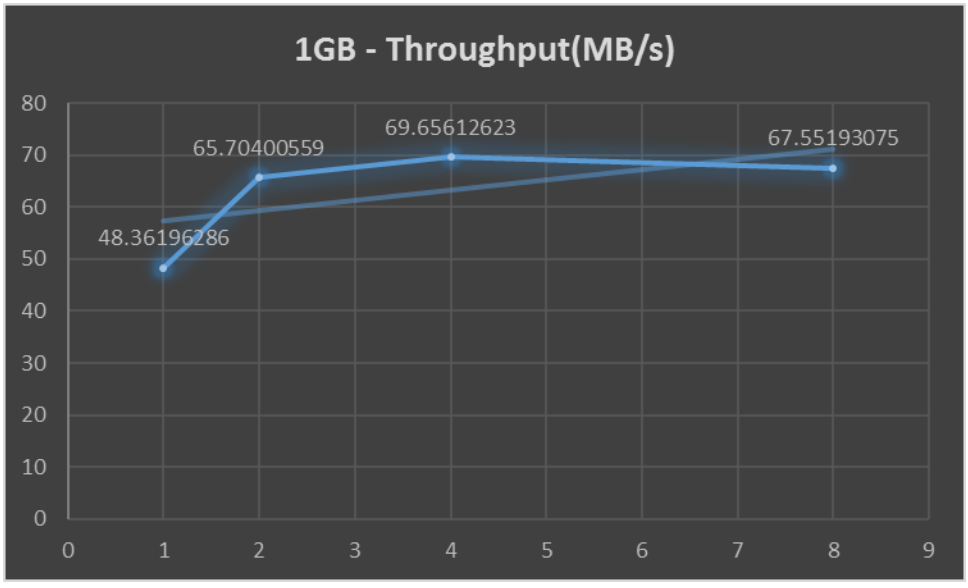
As you can see, it took me around 1 minute for entire sorting with multiple threads. It also takes low amount of time for file creation and validation.

Time (in secs)				
1GB-d2.xlarge	Thread 1	Thread 2	Thread 4	Thread 8
Gensort	11.46	11.46	11.46	11.46
Split	20	19	19	19
Quick	33	21	19	20
Merge	49	47	49	48
Valsort	9.6	9.6	9.6	9.6

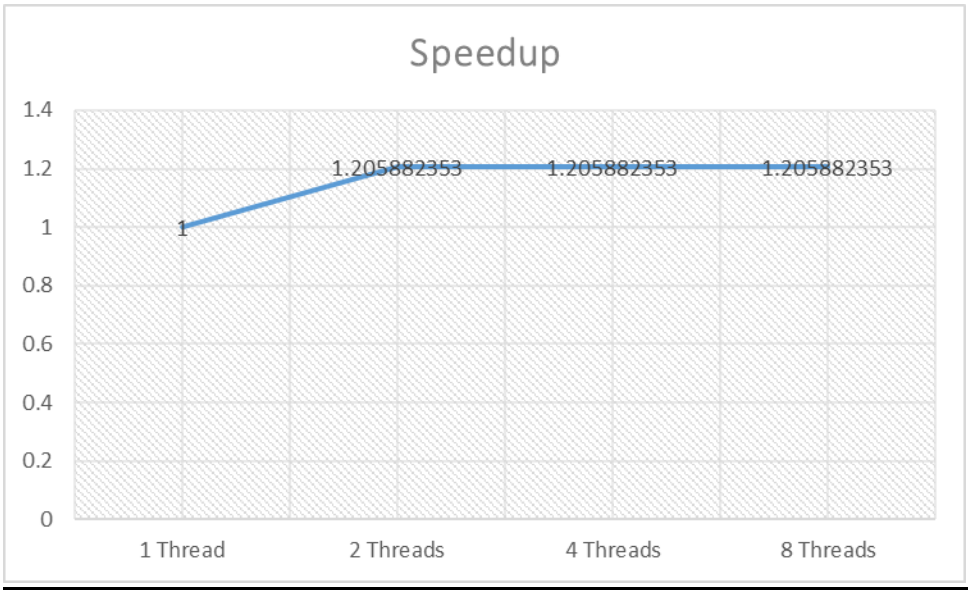


Throughput:

The throughput achieved with this instance starts from 48 MBps and goes upto 69MBps which is 20 MBps more than c3.large instance.



Speedup:



In case of speedup, it goes super linear with multiple threads in case of larger datasets but it remains sub linear though closer to 1 for low size datasets.

HadoopSort

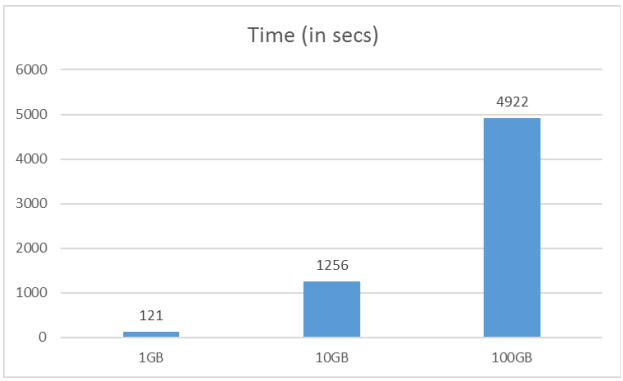
In this experiment, I implemented sorting algorithm using Hadoop's map and reduce programming model. In this model, everything is mapped to key-value pair and then sorted and reduced by key or value. In my case, I have considered each line of the file as a key and sorted according to the key. I ran 1GB and 10GB datasets on single node whereas for 100GB dataset I used 16 node cluster.

Time:

The time taken by this kind of programming model is more compared to my program for small datasets but it works much more efficiently for larger datasets which goes up to terabytes. In comparison to shared memory sort Hadoop takes less amount of time to sort a file of 10GB. The 100GB dataset was ran on 16 nodes of c3.large instance and it took around less than 2 hours to perform entire sort.

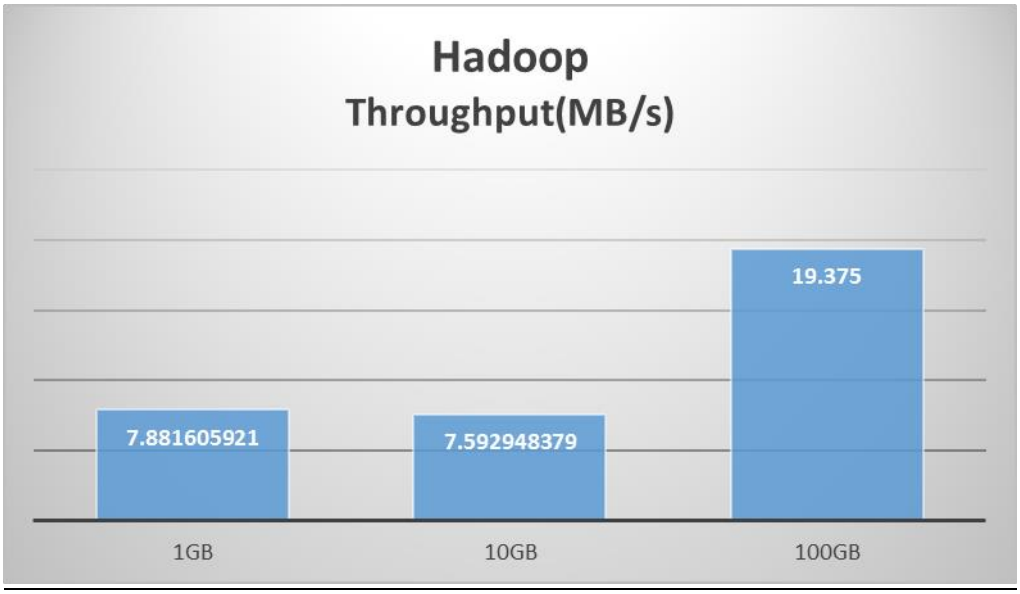
The next experiment I performed was on d2.xlarge instance of Amazon with 1GB dataset which took less amount of time than c3.large instance. The below table shows the time taken from generating the file until validation.

Time (in secs)				
	1GB	10GB	100GB	1GB -d2.xlarge
Gensort	14.76	161	1408	14.76
Sort	121	1256	4922	61
Valsort	14.09	163	804	11.77



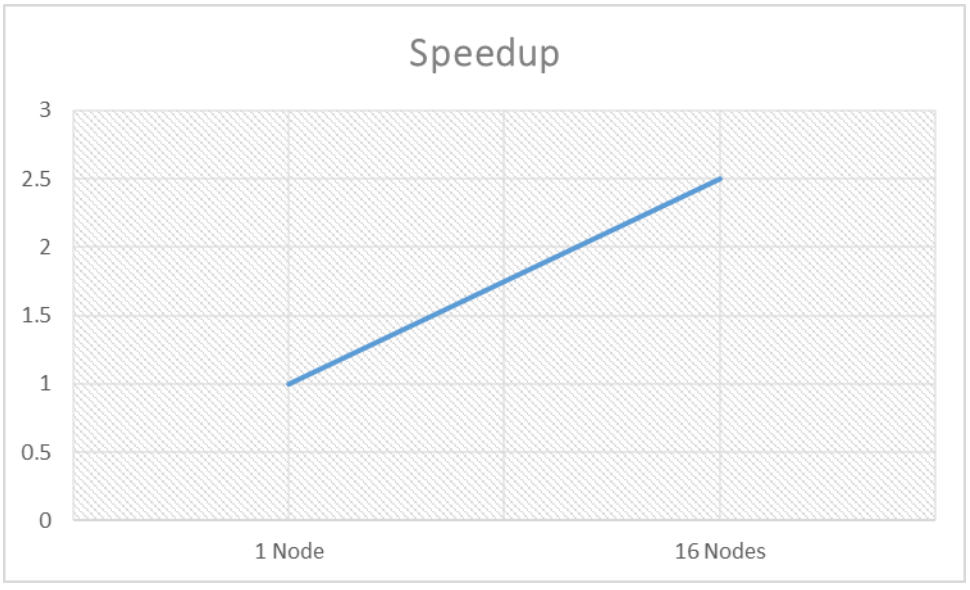
Throughput:

The below graph shows throughput achieved with different size of datasets in c3.large instance. For larger datasets such programming model gives optimum performance. Hence the throughput for 100GB is 19MBps and 10GB is only 7.6MBps which is lesser than Shared Memory.



Speedup:

Below graph shows that multi-node speedup goes superlinear. It is more than shared memory but lower than spark. For one node it is around 1 and for 16 nodes it is around 2.5.



SparkSort

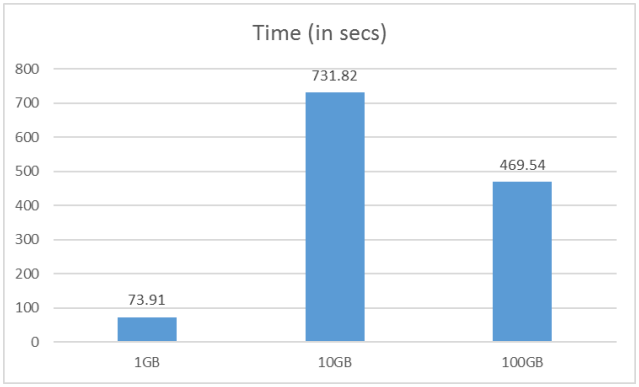
In this experiment, I used another programming model called as Spark which came after Hadoop with most of hadoop's loopholes fixed. This programming language also follows the same approach ad Hadoop but it works for real datasets which is iterative and dynamic. Hence I found it faster than Hadoop.

Time:

The below table depicts time taken by this programming model with datasets of different sizes. The 100GB dataset can be sorted only in 8 minutes whereas Hadoop takes around 2 hours. For 10GB dataset spark takes only 12 minutes compared to 20 minutes using Hadoop and around 20-30 minutes using Shared Memory. For 1GB also spark is better than Hadoop.

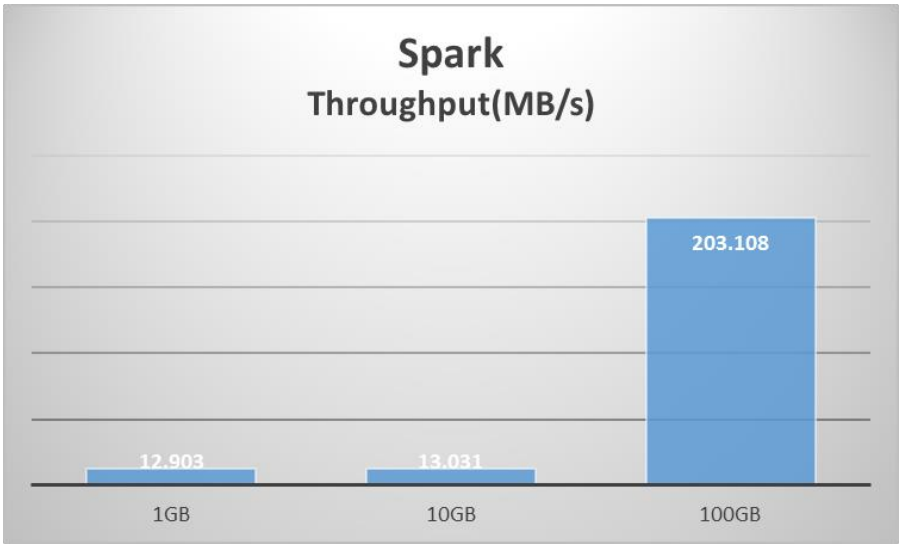
For next experiment, I ran spark for 1GB dataset on d2.xlarge instance which again takes less amount of time than c3.large instance.

Time (in secs)				
	1GB	10GB	100GB	1GB -d2.xlarge
Gensort	14.76	159	2040	9.7
Sort	73.91	731.82	469.54	44
Valsort	13.59	160	1342	10.05



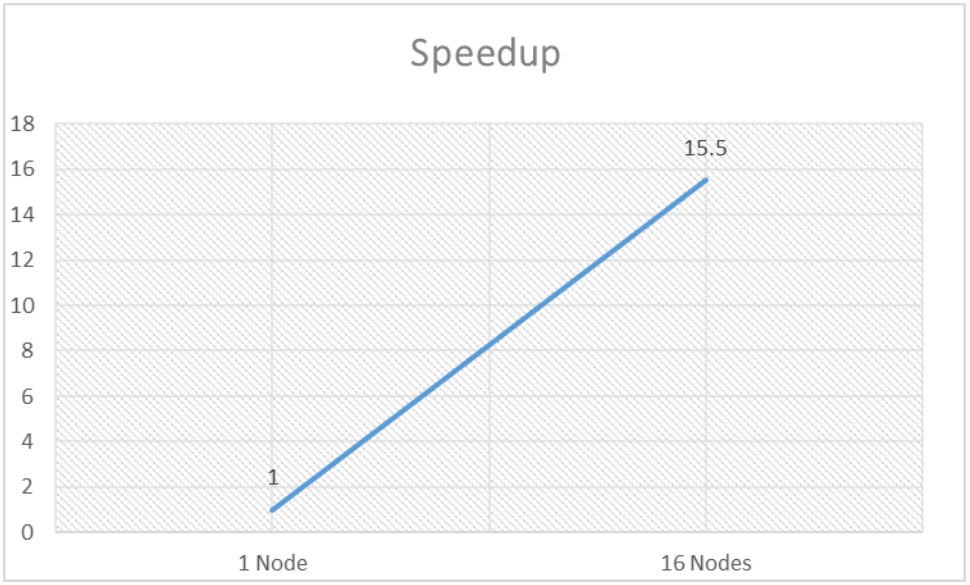
Throughput:

The below graph is throughput achieved using spark with different datasets on c3.large instance. As we can see, for datasets around 100GBs of size spark is way faster than Hadoop. Spark gives 200MBps on the other hand Hadoop gives only 20MBps. For 1GB and 10GB datasets also spark is faster than Hadoop.



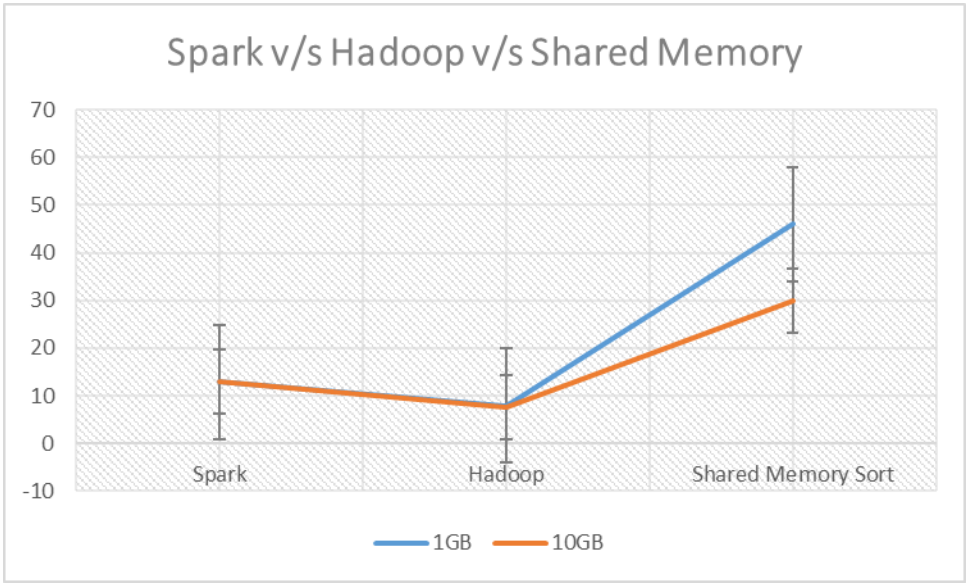
Speedup:

Below graph shows that multi-node speedup goes superlinear. It is more than both shared memory and Hadoop.

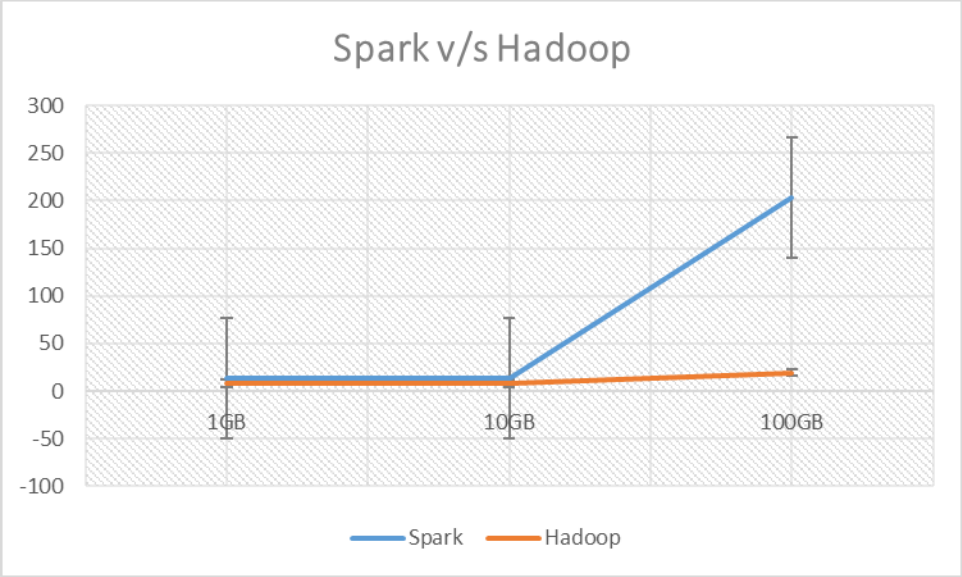


Comparisons:

In the below graph, I have compared the throughput for all the 3 sorting algorithms experimented for 1GB and 10GB datasets. I found that for small datasets of these size shared memory sort implemented using quick and merge sort is more efficient than both Hadoop and sort. Since the experiment for these datasets was carried on single node for Hadoop and spark I have used single threaded system of shared memory in the below graph.



The below graph is comparison of Hadoop and spark for 1GB, 10GB and 100GB datasets. Though 1GB and 10GB datasets were run on single node, the throughput will not exceed on 16 nodes for what I found for 100GB datasets. From the below graph, it is obvious that I found spark much faster than Hadoop. It requires less configuration manually and its iterative approach sorts larger datasets in minutes whereas Hadoop takes hours. Also, spark doesn't give a single merged file like Hadoop which might also be the reason for less sorting time.



Comparison Questions:

Q. What conclusions can you draw?

Answer: From the experiments performed in this assignment, I conclude that for small datasets shared memory sort seems to be faster than both Hadoop and Spark. For larger datasets starting from 100GBs Spark seems to be faster than Hadoop.

Q. Which seems to be best at 1 node scale?
Answer: Shared Memory Sort when ran on a larger instance with more no. of threads and process seems to me will work faster than Hadoop and spark for larger datasets also.

Q. How about 16 nodes?
Answer: From the above comparisons, it is obvious that I found Spark faster than Hadoop when ran on 16 nodes.

Q. Can you predict which would be best at 100 node scale?
Answer: I predict that for extremely larger datasets spark will give best performance on a 100 node scale.

Q. How about 1000 node scales?
Answer: Still, I would predict spark will give the best performance.

Q. Compare your results with those from the Sort Benchmark, specifically the winners in 2013 and 2014 who used Hadoop and Spark.

Answer:

As per system used in this assignment, c3.large instance of Amazon has only 3.75GB of RAM with 2 processors because of which we won't be able to store much data on main memory for computation.

Comparing the results with Winners of 2013, their Hadoop configuration includes 2100 nodes of 2 x 2.3GHz hexa-core processors with 64GB RAM. And hence the system was able to ran 100TB in the same amount of time c3.large used to sort 100GB on 16 nodes which is around 4500 seconds.

Comparing the results with Winners of 2014, their Spark configuration included 207 Amazon's i2.8xlarge nodes of 32 x 2.5GHz processors with 244GB of memory along with 6TB of storage. Hence their system was able to sort 100TB in 1400 seconds whereas in that amount of time c3.large instance will sort only 300GB of data.

Q. What can you learn from the CloudSort benchmark?

CloudSort Benchmark is sorting algorithm performed by Microsoft on Amazon EC2 and estimated the cost for sorting 100TB which will run for approx. 12 hours as $650. It uses 100 m1.large instances each with 840GB of storage and 3TBs of EBS. Each m1.large instance consists of 2 cores with 4 computational blocks and 7.5GB of memory. Hence the algorithm was able to sort 100TBs in 12 hours.

Solutions to the Questions:

conf/master

It contains the ip address of master. It is used by the slaves to connect to the master

conf/slaves

It contains the ip address of all the slaves. In a single node environment, it contains the ip address of that single node. In multi node, each slave will give its own ip address and master will contain ip-address of all the slaves

conf/core-site.xml

It contains the location of temp or intermediate files in Hadoop distributed file system

conf/hdfs-site.xml

It gives information about the Hadoop distributed file system such as replication, permissions, location of datanode and namenode.

conf/mapred-site.xml

It contains the host and port number where the mapreduce job tracker runs at. It also contains the details about mappers and reducers such as no. of mappers and reducers, memory for mappers and reducers.

Q1. What is a Master node? What is a Slaves node?
Answer:
Master node is the node which assigns handles the dataset and assigns workload to slaves. It also collects the final output data from all the slaves.
Slave node is the node which performs the actual computing work. It gets the data from the master, work on its share of data and submit the output data back to master.

Q2. Why do we need to set unique available ports to those configuration files on a shared environment? What errors or side-effects will show if we use same port number for each user?
Answer:
In a shared environment, there are different functions using different ports such as Master node, HDFS, and mapreduce job trackers. So, if we use same port for these functions, they will conflict with each other.

Q3. How can we change the number of mappers and reducers from the configuration file?
Answer:
We can change the number of mappers and reducers in the mapred-site.xml. The properties are "mapreduce.job.maps" and "mapreduce.jobs.reduce". "mapreduce.tasktracker.map.tasks.maximum" and "mapreduce.tasktracker.reduce.tasks.maximum" to change the maximum no. of reducers and mappers.

References:

http://spark.apache.org/docs/latest/index.html

http://hadoop.apache.org/

http://stackoverflow.com/

https://aws.amazon.com/documentation/ec2/

Conclusion:

From the above experiments, I found that spark is faster than Hadoop for datasets of sizes greater than 10GB. For smaller datasets simple shared memory sorting algorithms such as merge, quick are faster than map-reduce programming paradigms.

Note:

Deliverables: configuration files for single and multi-node given in configuration files folder, output files for all 3 sort given in output files, source code along with jar files are in source code folder.
Code.pdf, prog2_sourcecode.txt contains the code for all 3 sort algorithms, prog2_report.pdf contains the design and performance evaluation, Output Screenshots contains all the screenshots for various experiments, readme.txt contains steps to run the code and install.sh contains commands to install different software required for this assignment.