This programming assignment covers the TeraSort application implemented in 3 different ways: Java , Hadoop, and Spark.

CS553 Programming Assignment #2

Terasort Implemetation

Sumedha Gupta, Aditya Jadhav

Problem Description

- The problem is to implement the TeraSort application in 3 different ways: Java, Hadoop, and Spark.
- The sorting application should read a larger than memory files and sort it in place.
- Two datasets, a small and a large dataset used for sorting: 128GB dataset and 1TB dataset.
- Datasets have been generated using file generator "gensort".
- Datasets have been generated on demand every time to run the benchmark.

Methodology

- We have implemented our own external sort in java to accommodate the sorting in place of a larger than memory file.
    - We used ThreadPoolExecutor to facilitate multi-threading.
    - We first split the files and used TreeMap in java to sort them.
    - The code is implemented in such a way that the files are being split initially in to smaller files that can fit in to memory and sort is performed on the smaller splits using multiple threads. We used ThreadPool Executor to implement multithreading in java
    - Secondly, the smaller files are being merged concurrently by multiple threads.
- For Hadoop Map Reduce, Reducer and Mapper classes have been implemented for sorting the files.
    - This is implemented using Hadoop MapReduce in two parts. First is the mapper which takes each line and split that in to key and value. These intermediate key value pairs get sorted by key by the MapReduce framework automatically and sends the pair further to the reducer. This phase of transferring the mapper output to the reducer is called as shuffling and it can take place before the completion of map phase and results in a better execution time.
    - Second is the reducer which takes the key value from the mapper and output a key value pair in return.
- For Spark, spark sorting is used to sort the files.
    - This implementation is performed using Scala. As Scala does not have its own file system and relies on Hadoop HDFS. So, it takes the input from the hdfs.

Runtime Environment Settings

Note: We have used Maven- 3.3.9 instead of ANT

- Amazon Linux AMI 2017.09.1
- Java: 1.8
- Hadoop: Apache Hadoop 2.9
- Spark: Apache Spark 2.2.0

Installation steps to setup your virtual cluster

1. Setup of i3.large 1 node:
   Launch a instance of type i3.large on AWS EC2 and run the following
   a. SSH setup:
      - ◊ scp -i <key> config <namenode>:~/.ssh/
      - ◊ scp -i <key> <pemkey> <namenode>:~/.ssh/
      - ◊
   b. RAID 0 setup and disk mounting on the OS.
      - ◊ *Reference- http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/raid-config.html*
      - ◊ sudo mkfs.ext4 /dev/nvme0n1
      - ◊ sudo mke2fs -F -t ext4 /dev/nvme0n1

      - ◊ sudo mkfs.ext4 /dev/nvme1n1
      - ◊ sudo mke2fs -F -t ext4 /dev/nvme1n1
      - ◊ sudo mdadm --create --verbose /dev/md0 --level=0 --name=MY_RAID --raid-devices=2 /dev/nvme0n1 /dev/nvme1n1
      - ◊ sudo mkfs.ext4 -L MY_RAID /dev/md0
      - ◊ sudo mkdir -p /raid
      - ◊ sudo mount LABEL=MY_RAID /raid
   c. Hadoop Installation
      - ◊ sudo yum update
      - ◊ sudo yum install java-1.8.0
      - ◊
      - ◊ ssh-keygen -f ~/.ssh/id_rsa -t rsa -P ""
      - ◊ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
      - ◊
      - ◊ cd ~
      - ◊ wget http://mirrors.koehn.com/apache/hadoop/common/hadoop-2.9.0/hadoop-2.9.0.tar.gz
      - ◊ tar -zxvf hadoop-2.9.0.tar.gz
      - ◊ chmod 777 hadoop-2.9.0
      - ◊ exit 0

2. Setup of i3.4xlarge 1 node:
   Launch a instance of type i3.4xlarge on AWS EC2 and run the following
   a. SSH setup:
      - ◊ scp -i <key> config <namenode>:~/.ssh/
      - ◊ scp -i <key> <pemkey> <namenode>:~/.ssh/
      - ◊
   b. RAID 0 setup and disk mounting on the OS.
      - ◊ *Reference- http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/raid-config.html*
      - ◊ sudo mkfs.ext4 /dev/nvme0n1
      - ◊ sudo mke2fs -F -t ext4 /dev/nvme0n1

      - ◊ sudo mkfs.ext4 /dev/nvme1n1

- ◊ sudo mke2fs -F -t ext4 /dev/nvme1n1
- ◊ sudo mdadm --create --verbose /dev/md0 --level=0 --name=MY_RAID --raid-devices=2 /dev/nvme0n1 /dev/nvme1n1
- ◊ sudo mkfs.ext4 -L MY_RAID /dev/md0
- ◊ sudo mkdir -p /raid
- ◊ sudo mount LABEL=MY_RAID /raid

c. Hadoop Installation

- ◊ sudo yum update
- ◊ sudo yum install java-1.8.0
- ◊
- ◊ ssh-keygen -f ~/.ssh/id_rsa -t rsa -P ""
- ◊ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
- ◊
- ◊ cd ~
- ◊ wget http://mirrors.koehn.com/apache/hadoop/common/hadoop-2.9.0/hadoop-2.9.0.tar.gz
- ◊ tar -zxvf hadoop-2.9.0.tar.gz
- ◊ chmod 777 hadoop-2.9.0
- ◊ exit 0

3. Setup of i3.large 8 nodes:
   Launch 8 instances of i3.large on Amazon EC2 and name the instances as follows:
   - ◊ NameNode
   - ◊ DataNode1
   - ◊ DataNode2
   - ◊ DataNode3
   - ◊ DataNode4
   - ◊ DataNode5
   - ◊ DataNode6
   - ◊ DataNode7

a. Hadoop Installation on all the nodes:

- ◊ sudo yum update
- ◊ sudo yum install java-1.8.0
- ◊
- ◊ ssh-keygen -f ~/.ssh/id_rsa -t rsa -P ""
- ◊ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
- ◊
- ◊ cd ~
- ◊ wget http://mirrors.koehn.com/apache/hadoop/common/hadoop-2.9.0/hadoop-2.9.0.tar.gz
- ◊ tar -zxvf hadoop-2.9.0.tar.gz
- ◊ chmod 777 hadoop-2.9.0
- ◊ exit 0

b. Setup environment variables as follows:
JAVA_HOME= / usr/lib/jvm/java-1.7.0-openjdk-1.7.0.151.x86_64

c. Update core_site.xml on all the nodes as follows:

```
<configuration>
<property>
<name>fs.defaultFS</name>
<value><nnode>:9000</value>
</property>
</configuration>
```

d. Create data directory on all nodes:
```
sudo mkdir -p /usr/local/hadoop/hdfs/data
```
e. Configure namenode and update the authorized keys on all datanodes
```
namenode> ssh-keygen
cat id_rsa.pub >> ~/.ssh/authorized_keys
```
f. Set up ssh.config on namenode and now you will be able to access all datanodes from namenode.
g. Set HDFS properties and mapreduce properties as follows:
```
<configuration>
 <property>
  <name>dfs.replication</name>
  <value>3</value>
 </property>
 <property>
  <name>dfs.namenode.name.dir</name>
  <value>file:///usr/local/hadoop/hdfs/data</value>
 </property>
</configuration>

<configuration>
 <property>
  <name>mapreduce.jobtracker.address</name>
  <value><nnode>:54311</value>
 </property>
 <property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
 </property>
</configuration>
```

h. Setup master and slaves and configure datanodes.
i. Finally, start the cluster as follows:
```
namenode> ./hadoop-2.7.3/sbin/start-dfs.sh
namenode> ./hadoop-2.7.3/sbin/start-yarn.sh
namenode> ./hadoop-2.7.3/sbin/mr-jobhistory-daemon.sh start historyserver
```

Difficulties in setup of the virtual cluster

- We faced issues for the setup of Hadoop on the cluster and the distribution of memory for mapper and reducer.
- Also, we faced the issue of unhealthy node due to limited space.
- We faced issues in password less ssh setup.
- There were issues faced for the setup of 8 nodes cluster on Hadoop.
- There were issues faced for GC overhead limit execeeded.

Versions Used

- Amazon Linux AMI 2017.09.1
- Java: 1.8
- Maven- 3.3.9
- Hadoop: Apache Hadoop 2.9
- Spark: Apache Spark 2.2.0

Performance Comparison

Performance: Compare the performance of the four versions of TeraSort (Shared-Memory, Hadoop, Spark, and MPI) on 1 node scale and 8 node scale with both the 128GB and 1TB dataset; Your time should be reported in seconds. Some of the things that will be interesting to explain are:

- How many threads, mappers, reducers, you used in each experiment
  - We used 8 threads for 1 TB data and 4 threads for 128GB data in case of shared memory.
  - We used 1024 mapper for 128 GB and 4 reducers, for 1 TB there were 8192 mapper and 8 reducers.

- How many times did you have to read and write the dataset for each experiment.
  For shared memory- 3 Data Reads and 3 Data Writes.
  For Hadoop and Spark, 2 Data Reads and 2 Data Writes

- What speedup and efficiency did you achieve (use weak scaling)?
  For Hadoop, we achieved a speed up of 3.26 and 5 for Spark.
  Efficiency is 65.4 and 71.6 for Hadoop and Spark respectively.

Compare the performance of the three versions of TeraSort (Shared-Memory, Hadoop, Spark, and MPI [EC]) on 1 node scale on two different types of instances as well as on a virtual cluster of 8 nodes, and explain your observations; compare the Shared-Memory performance of 1 node to Hadoop and Spark TeraSort at 8 node scales and explain your observations. You should be doing weak scaling experiments as you scale up from 1 node to 8 nodes. You only need to do two different scales, 1 node and 8 nodes (no need to incrementally do 1, 2, 4, and 8).

Throughput has been calculated as: ***Total Data Read + Write in MB/ Compute Time in secs***

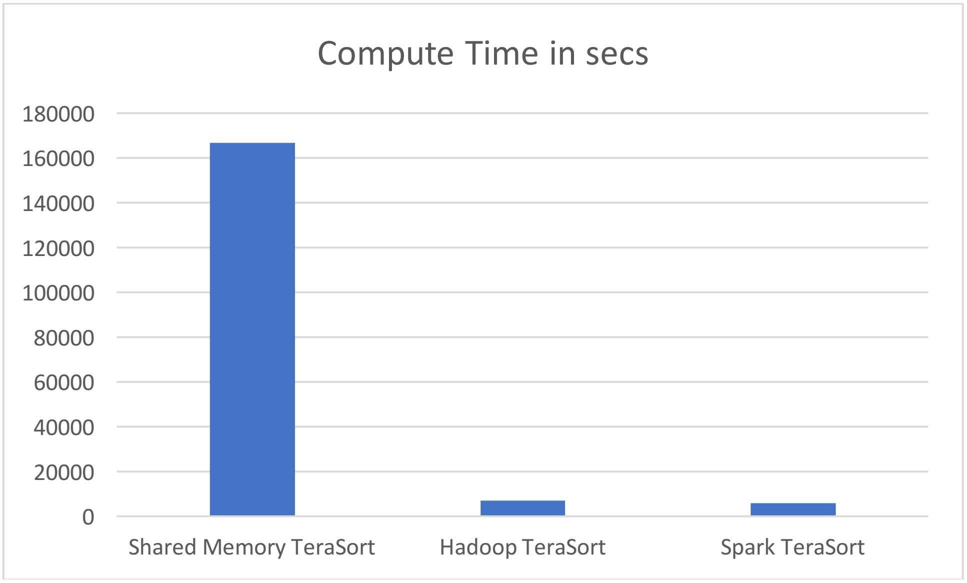
Speed-Up = 8 * time for 128GB/ time for 1TB (between i3.4xlarge and i3.large)

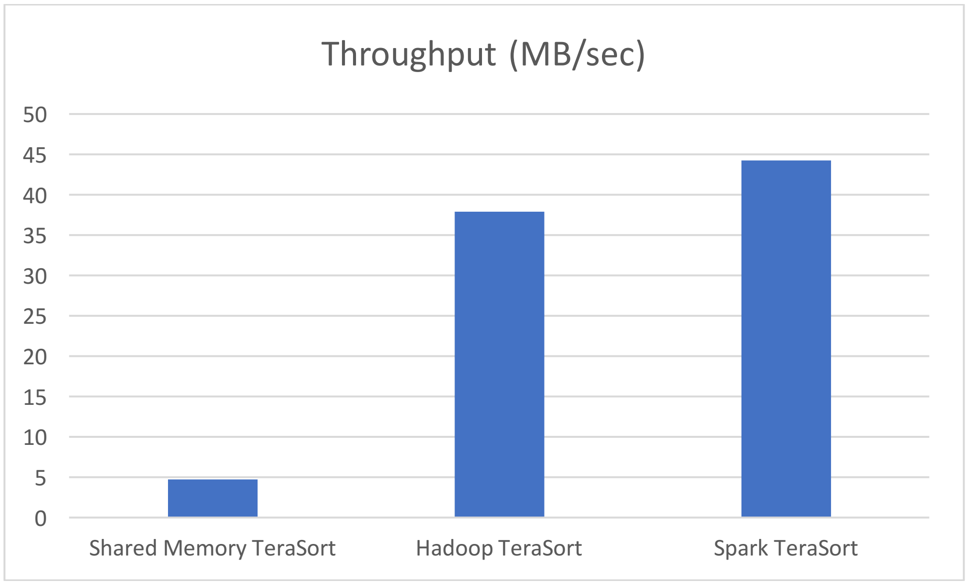
Efficiency= 100 * time for 128GB/ time for 1TB %

Experiment (instance/dataset)	Shared Memory Terasort	TeraSort Hadoop	TeraSort Spark
Compute Time (sec) [1x i3.large 128GB]	166700.49	6914.499	5928.423
Data Read (GB) [1x i3.large 128GB]	384GB	128GB	128GB
Data Write (GB) [1x i3.large 128GB]	384GB	128GB	128GB
I/O Throughput (MB/sec) [1x i3.large 128GB]	4.717	37.912	44.22
Compute Time (sec) [1x i3. 4x large 1 TB]	31580.442	19108.543	8543.6
Data Read (GB) [1x i3. 4x large 1 TB]	3072	1024	1024
Data Write (GB) [1x i3.4xlarge 1 TB]	3072	1024	1024
I/O Throughput (MB/sec) [1x i3.4xlarge 1 TB]	19.922	50.76	243.4
Compute Time (sec) [8x i3. large 1TB]	N/A	17690.64	8267.46
Data Read (GB) [8x i3. large 1TB]	N/A	1024	1024
Data Write (GB) [8x i3. large 1TB]	N/A	1024	1024
I/O Throughput (MB/sec) [8x i3. large 1TB]	N/A	69.85	253.7
Speedup (weak scale)	1.22	3.36	5.55
Efficiency (weak scale)	52.78	65.4	71.62

Performance Comparison Graphs

1. Comparison of the performance of the TeraSort for Shared-Memory, Hadoop, Spark on 1 node scale with 128GB dataset (**i3.large**)

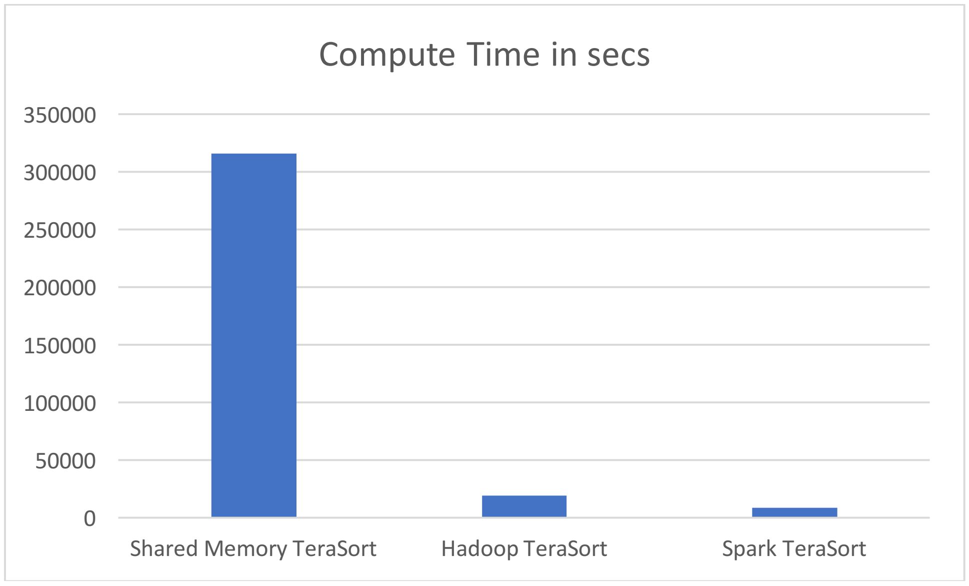


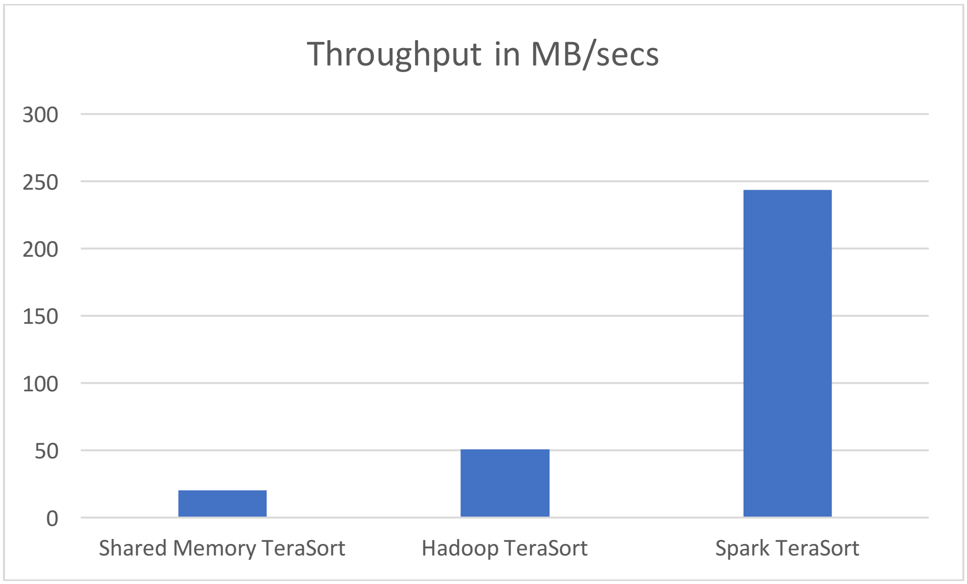


Explanation and Conclusion:

The compute time and throughput of Spark is the best. The shared memory takes the highest time out of all the three configurations.

2. Comparison of the performance of the TeraSort for Shared-Memory, Hadoop, Spark on 1 node scale with 1TB dataset (**i3.4xlarge**)

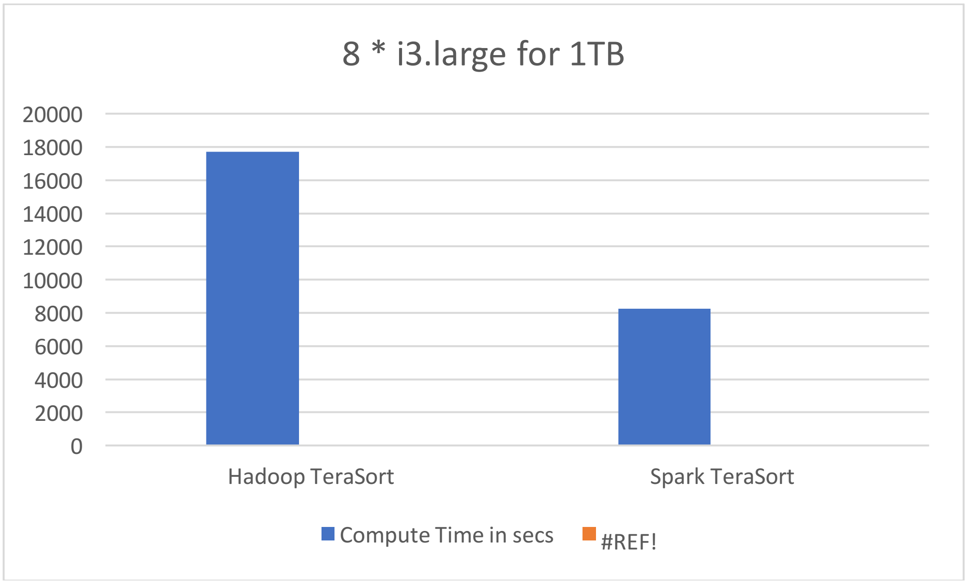


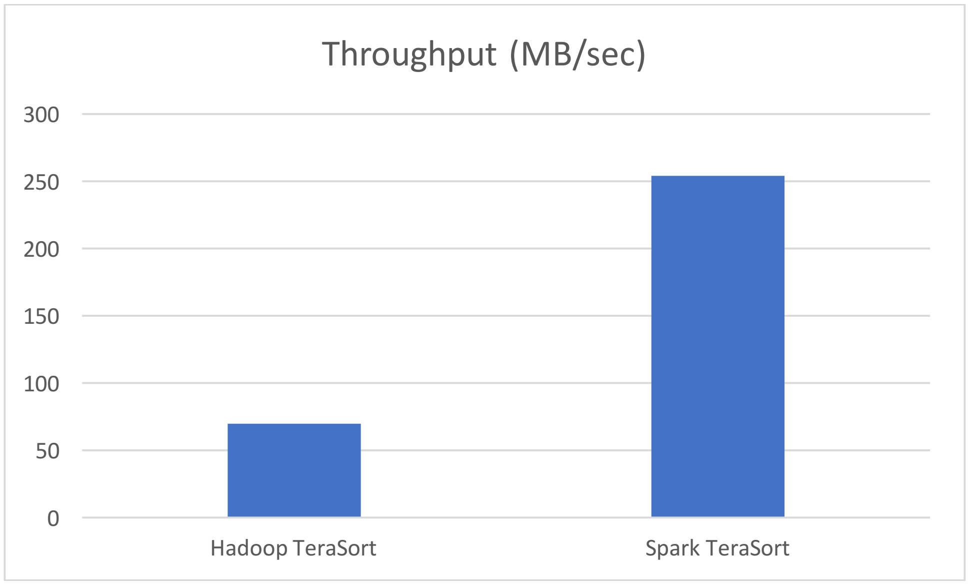


Explanation and Conclusion:

The compute time and throughput of Spark is the best. The shared memory takes the highest time out of all the three configurations.

3. Comparison of the performance of the of TeraSort for Hadoop, Spark on 8 nodes scale with 1TB dataset (**i3.large**)

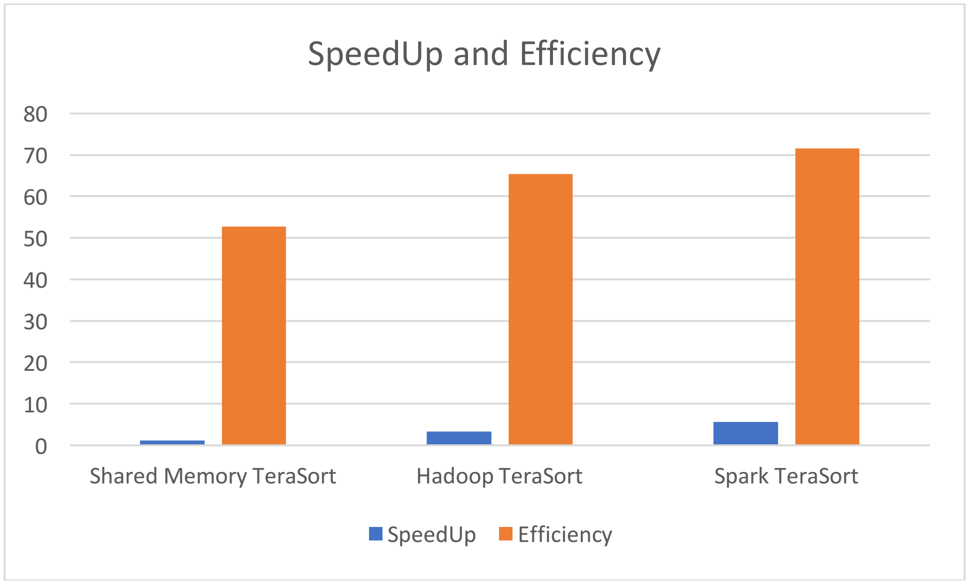




EXPLANATION AND CONCLUSION:

In this, Spark is much better than Hadoop in terms of time and throughout.

4. Comparison the speedup and efficiency of the of TeraSort for Shared-Memory, Hadoop, Spark.



EXPLANATION AND CONCLUSION:

The speed up of Spark is the best as it is highly scalable. Also, in the terms of efficiency it is the best.

- Which seems to be best at 1 node scale? How about 8 nodes?

1. Spark seems to be the best for 1 node scale as its performance is the best in terms of compute time. This is because Spark uses in memory sorting and that's why it is very fast. It seems to be much better than shared memory and a little better than Hadoop.
2. For 8 nodes cluster, Spark performs the best as well.

- Can you predict which would be best at 100 nodes scale? How about 1000 node scales?

If we see the graphs for speed up and efficiency, we can observe that the speed up of Spark is increasing as the number of nodes increasing. Thus, we can say that it is highly scalable and utilize the resource as much as possible as the scale increases. Hence, it will perform better for 100 and 1000 nodes scale as well.

- Compare your results with those from the Sort Benchmark [9], specifically the winners in 2013 and 2014 who used Hadoop and Spark.

The winners in 2013 and 2014 who used Hadoop and Spark are triton Sort and Baidu Sort. If we relate the performance with our experiment we can say that Spark was best in their experiment as well and in ours also with high compute time and throughput which is due to the fact that Spark is highly scalable and utilize resources efficiently.

- What can you learn from the CloudSort benchmark, a report can be found at [10].

  - Cloud Sort provides us the efficiency of external sort in terms of Total Cost of Ownership.
  - Cloud Sort provides us the cost to sort a fixed number of records on the public cloud.
  - It can also tell us the best platforms for building data pumps.
  - It can also tell the most efficient sort implementations in terms of Total Cost of Ownership.