PROGRAMMING ASSIGNMENT 2 REPORT

Kunal Dhande
kdhande@hawk.iit.edu

SHARED MEMORY

1. DESCRIPTION

It is sorting program to sort record of 100 bytes with large dataset, where memory is less compared to data size with multi-threading (1, 2, 4, and 8 threads) on 1 virtual node.

2. METHODOLOGY

The program divides given dataset into chunks of size 1MB and then perform merge sort on it. Each record is of size 100 bytes has key with size 10 bytes and 88 bytes value. Later, it merges all sorted files into one file.

3. RUNTIME ENVIRONMENT

AWS instance: d2.xlarge (Ubuntu OS 14.04)
Data set size: 1 GB and 1 TB
Virtual nodes: 1 node
Language of implementation: C (gcc compiler)

HADOOP

1. DESCRIPTION

It is sorting program to sort record of 100 bytes with large dataset by using Hadoop Map-Reduce model.

2. METHODOLOGY

In this program, sorting is implemented by using inbuilt soring functionality by Hadoop using Map-Reduce model. In Map function, we are dividing each record into value and key. Each record is of size 100 bytes has key with size 10 bytes and 88 bytes value. While execution, after completing mapping, there is shuffle phase, there Hadoop it self sort data with respect to key.

3. RUNTIME ENVIRONMENT

AWS instance: d2.xlarge (Ubuntu OS 14.04)
Hadoop Version: Hadoop 2.7.2
Data set size: 1 GB and 1 TB
Virtual Nodes: 1 node and 17 nodes (1 master + 16 slaves)
Language of implementation: Java (Version: 1.8.0_77)

4. CONFIGURATION FILES

1. conf/slaves:

Slaves contain a rundown of hosts, one for every line, that are expected to have DataNode and TaskTracker servers.

Single node:

```
localhost
```

Multi-node:

```
HadoopMaster
HadoopSlave1
HadoopSlave2
HadoopSlave3
HadoopSlave4
```

```
HadoopSlave5
HadoopSlave6
HadoopSlave7
HadoopSlave8
HadoopSlave9
HadoopSlave10
HadoopSlave11
HadoopSlave12
HadoopSlave13
HadoopSlave14
HadoopSlave15
HadoopSlave16
```

2. conf/master:

The Masters contain a rundown of hosts, one for each line, that are required to host auxiliary NameNode servers. The Masters document illuminates about the Secondary NameNode area to Hadoop daemon.

Single node:

```
localhost
```

Multi-node:

```
HadoopMaster
```

3. conf/core-site.xml

The core-site.xml file informs Hadoop daemon where NameNode runs in the cluster. It contains the configuration settings for Hadoop Core such as I/O settings that are common to HDFS and Map-Reduce.

Single node:

```
<property>
     <name>fs.default.name</name>
     <value>hdfs://HadoopMaster:9000</value>
</property>
<property>
     <name>hadoop.tmp.dir</name>
     <value>/mnt/raid/tmp/</value>
</property>
```

Multi-node:

```
<property>
     <name>fs.default.name</name>
     <value>hdfs://localhost:9000</value>
</property>
<property>
     <name>hadoop.tmp.dir</name>
     <value>/mnt/raid/tmp/</value>
</property>
```

4. conf/hdfs-site.xml

The hdfs-site.xml file contains the configuration settings for HDFS daemons; the NameNode, the Secondary NameNode, and the DataNodes. Here, we can configure hdfs-site.xml to specify default block replication and permission checking on HDFS. The actual number of replications can also be specified when the file is created. The default is used if replication is not specified in create time.

```
<property>
     <name>dfs.replication</name>
     <value>1</value>
</property>
<property>
     <name>dfs.namenode.name.dir</name>
     <value>/mnt/raid/hdfs/namenode</value>
</property>
<property>
     <name>dfs.datanode.data.dir</name>
     <value>/mnt/raid/hdfs/datanode</value>
</property>
```

5. conf/mapred-site.xml

The mapred-site.xml file contains the configuration settings for Map Reduce daemons; the job tracker and the task-trackers.

5. CONCEPTS:

1. What is Master Node? What is Slave Node?

Master node is the node which controls other nodes to execute number of operations Slave node is the node which operates under master node to perform various operations. This configuration is basically used for load sharing purposes and faster execution.

2. 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?

If ports are different across number of nodes, then it becomes difficult for master to manage various operations. And unique ports manage different operations

3. How can we change the number of mappers and reducers from the configuration file?

To change number of mappers and reducers, we need to add some configuration settings to yarn-site.xml

```
<property>
     <name>map.cpu.vcores</name>
     <value>4</value>
</property>
<property>
     <name>reduce.cpu.vcores</name>
     <value>4</value>
</property>
```

SPARK

1. DESCRIPTION

It is sorting program to sort record of 100 bytes with large dataset by using Spark Map-Reduce model.

2. METHODOLOGY

In this program, sorting is implemented by using inbuilt soring functionality by Spark using Map-Reduce model. In Map function, we are dividing each record into value and key. Each record is of size 100 bytes has key with size 10 bytes and 88 bytes value. While execution, after completing mapping, there is shuffle phase, there Hadoop it self sort data with respect to key.

3. RUNTIME ENVIRONMENT

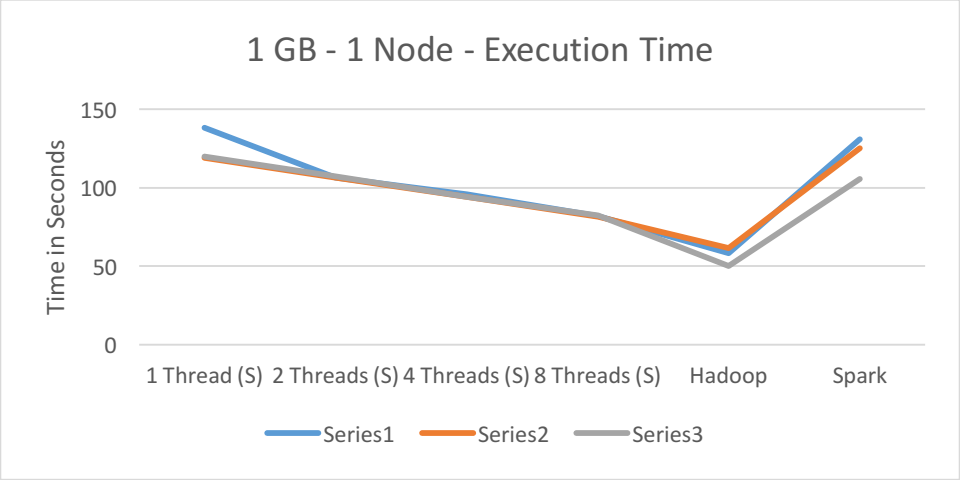
AWS instance: d2.xlarge (Ubuntu OS 14.04)
Spark Version: Spark 1.6.1 pre-built for Hadoop 2.6 and higher
Data set size: 1 GB and 1 TB
Virtual Nodes: 1 node and 17 nodes (1 master + 16 slaves)
Language of implementation: Python (Version 2.7)

PERFORMANCE

1. Single Node

1. DATASET: 1 GB

Total Time (Sec.)			
Sorting Strategy	**Run 1**	**Run 2**	**Run 3**
1 Thread (Shared Memory)	138.301	119.123	120.032
2 Threads (Shared Memory)	106.518	106.453	107.325
4 Threads (Shared Memory)	95.871	94.245	94.025
8 Threads (Shared Memory)	81.809	81.463	82.534
Hadoop	58.312	61.53	50.32
Spark	131.089	125.32	105.43

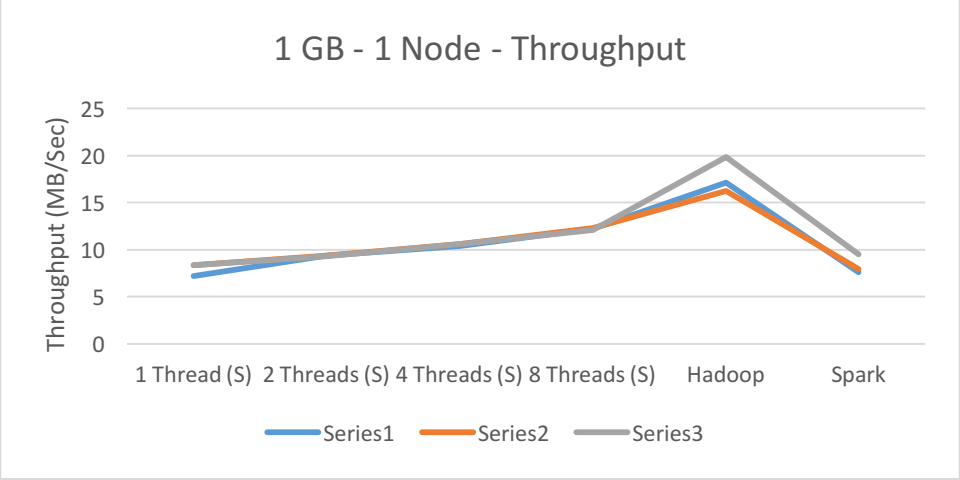


(S) REPRESENTS: SHARED MEMOTY

To sort 1 GB data on single node, Hadoop completes its sorting faster whereas Shared Memory and spark has almost same speed.

Throughput (MB/Sec)			
Sorting Strategy	**Run 1**	**Run 2**	**Run 3**
1 Thread (Shared Memory)	7.230605708	8.394684486	8.331111704
2 Threads (Shared Memory)	9.388084643	9.39381699	9.317493594
4 Threads (Shared Memory)	10.4306829	10.61064247	10.63546929
8 Threads (Shared Memory)	12.22359398	12.27551158	12.11621877
Hadoop	17.14912882	16.25223468	19.87281399
Spark	7.628405129	7.979572295	9.484966328

While sorting 1 GB data on single node, Hadoop gives more throughput whereas shared memory gives lowest throughput among them.

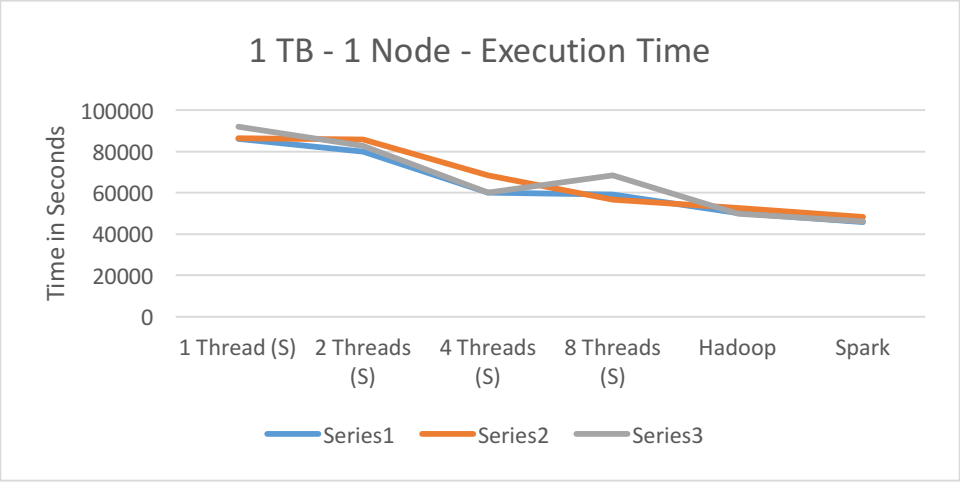


(S) REPRESENTS: SHARED MEMOTY

Winner: Hadoop

2. DATASET: 1 TB

Total Time (Sec.)			
Sorting Strategy	**Run 1**	**Run 2**	**Run 3**
1 Thread (Shared Memory)	86132.345	86410.534	91890.765
2 Threads (Shared Memory)	79881.563	85890.765	82592.242
4 Threads (Shared Memory)	60141.125	68552.125	60234.836
8 Threads (Shared Memory)	59225.876	56608.836	68476.436
Hadoop	50275.36	52643.643	49754.765
Spark	45923.342	48143.532	46243.243

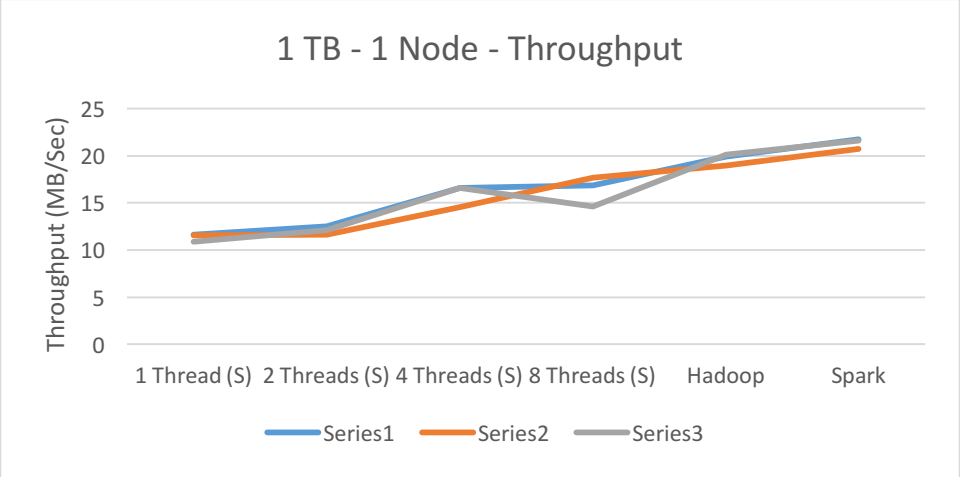


(S) REPRESENTS: SHARED MEMOTY

To sort 1 TB data on single node, Spark completes sorting faster, whereas shared memory with 1 thread is slowest among them.

Throughput (MB/Sec)			
Sorting Strategy	**Run 1**	**Run 2**	**Run 3**
1 Thread (Shared Memory)	11.61004034	11.57266312	10.8824864
2 Threads (Shared Memory)	12.51853322	11.64269523	12.10767471
4 Threads (Shared Memory)	16.62755727	14.58743985	16.60168876
8 Threads (Shared Memory)	16.88451176	17.66508677	14.60356377
Hadoop	19.89045926	18.99564588	20.09857749
Spark	21.7754187	20.77122219	21.62478094

While sorting 1 TB data on single node, Spark gives more throughput whereas shared memory gives lowest throughput among them.



(S) REPRESENTS: SHARED MEMOTY

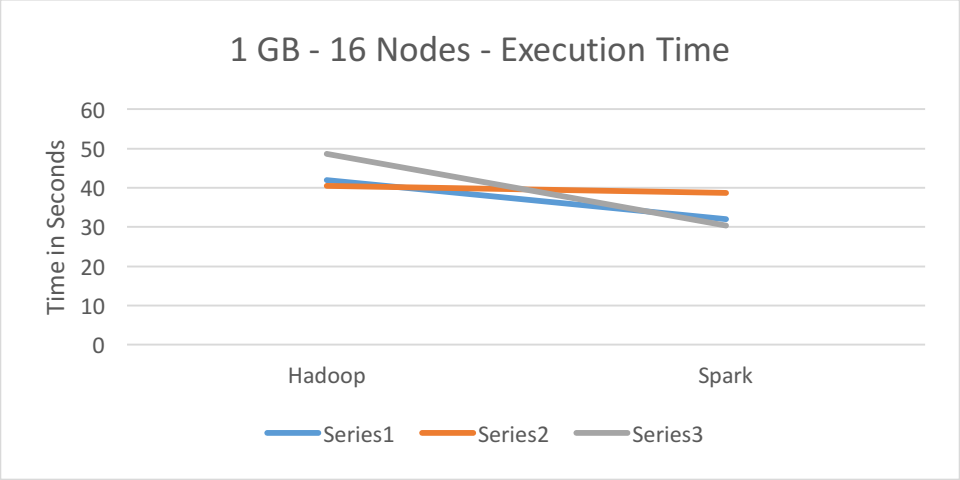
Winner: Spark

2. Multi Node

1. DATASET: 1 GB

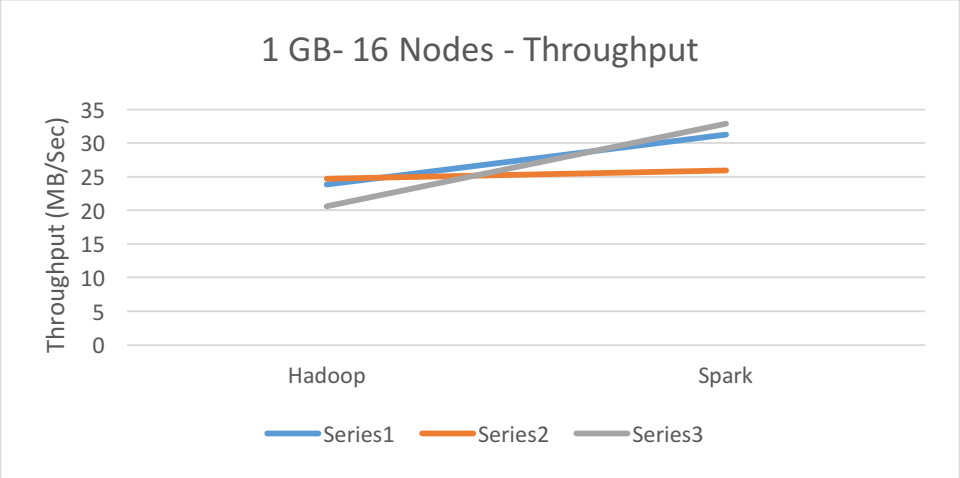
Total Time (Sec.)			
Sorting Strategy	**Run1**	**Run2**	**Run3**
Hadoop	42	40.436	48.643
Spark	32.013	38.644	30.423

To sort 1 GB data on 16 nodes, Spark completes its sorting faster than Hadoop.



Throughput (MB/Sec.)			
Sorting Strategy	**Run1**	**Run2**	**Run3**
Hadoop	23.80952381	24.73043822	20.55794256
Spark	31.23730984	25.87723838	32.86986819

While sorting 1 GB data on 16 nodes, spark gives more throughput than Hadoop.

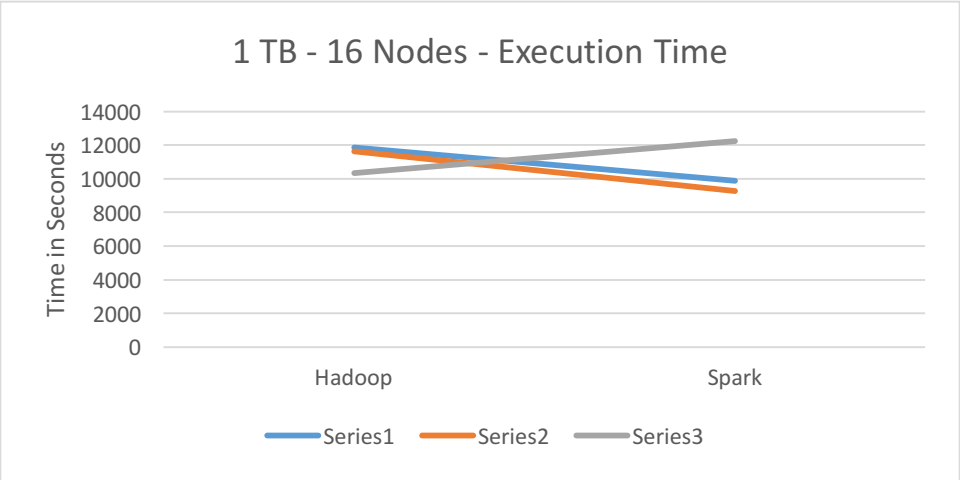


Winner: Spark

2. DATASET: 1 TB

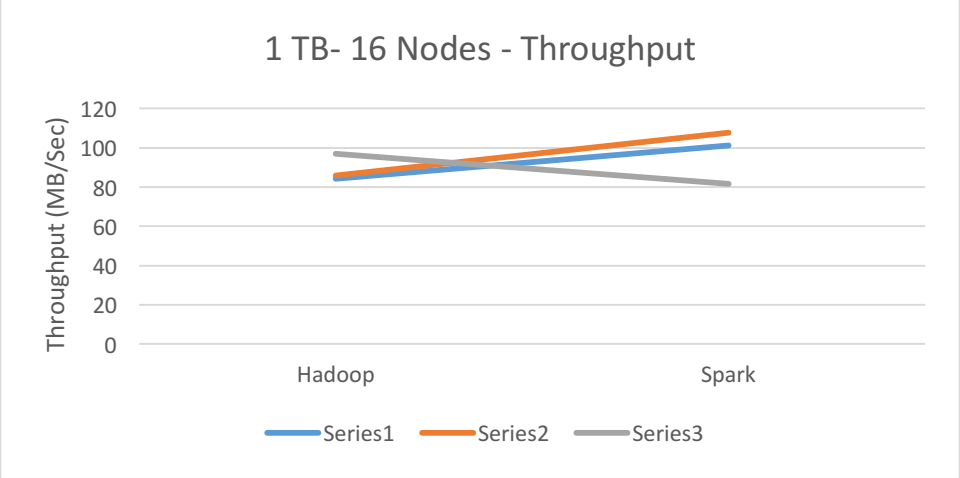
Total Time (Sec.)			
Sorting Strategy	**Run1**	**Run2**	**Run3**
Hadoop	11873.104	11636.345	10323.547
Spark	9883.435	9283.765	12223.646

To sort 1 TB data on 16 nodes, Spark has better performance compared to Hadoop.



Throughput (MB/Sec.)			
Sorting Strategy	**Run1**	**Run2**	**Run3**
Hadoop	84.22397378	85.93763763	96.86593184
Spark	101.1793976	107.7149195	81.80865185

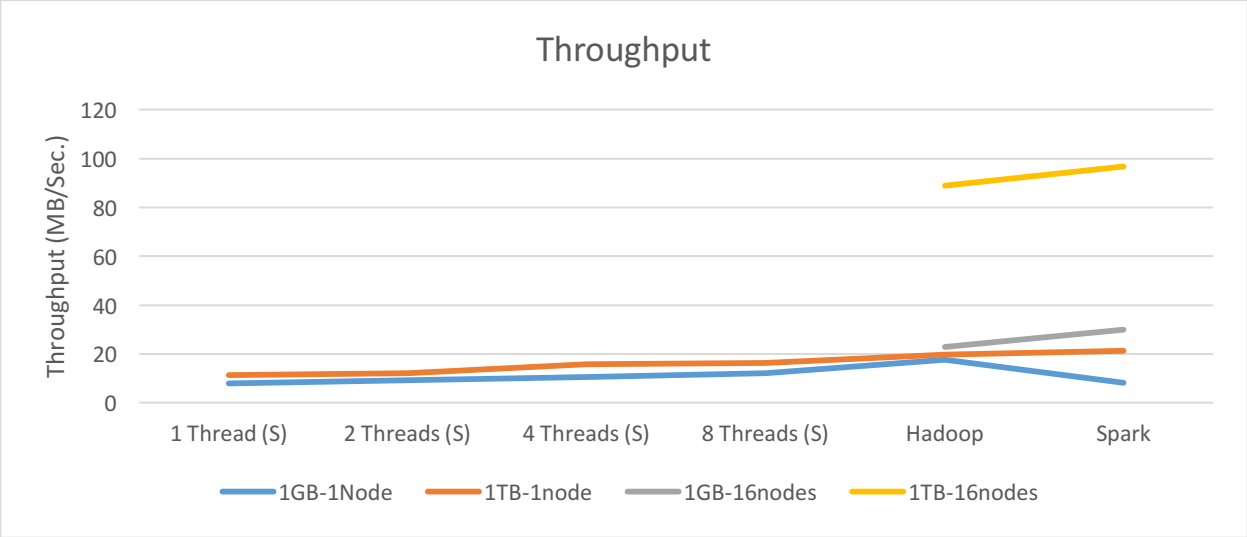
While sorting 1 TB data on 16 nodes, Spark gives more throughput Than Hadoop.



Winner: Spark

AVERAGE THROUGHPUT:

Throughput (MB/Sec)				
	1 Node		16 Nodes	
	1 GB	1 TB	1 GB	1 TB
1 Thread (Shared Memory)	7.985467299	11.35506329	-	-
2 Threads (Shared Memory)	9.366465076	12.08963439	-	-
4 Threads (Shared Memory)	10.55893155	15.93889529	-	-
8 Threads (Shared Memory)	12.20510811	16.38438744	-	-
Hadoop	17.75805917	19.66156088	23.03263486	89.00918108
Spark	8.364314584	21.39047394	29.99480547	96.90098968



In above graph, you can say that shared memory has lowest throughput, but it increases as number of threads increases. Hadoop gives best throughput when data size is less. But it has moderate throughput. Whereas, spark has exceptional performance on large data set and multi node.

Overall Winner: Spark

QUESTIONS

1. WHICH SEEMS TO BE BEST AT 1 NODE SCALE?

For 1 node, Hadoop seems to be best. It has better throughput than shared Memory and Spark for 1 node. Spark give better performance for large datasets, but as size of dataset decreases spark show huge difference in throughput.

2. HOW ABOUT 16 NODES?

For 16 nodes, spark is fastest. It goes on increasing throughput as you add nodes. Spark is highly scalable. Spark is much useful than Hadoop if number of nodes and data size is big.

3. OVERALL

Overall we can say that spark is better. Although for smaller data sets spark lacks behind Hadoop, but still we use such kind of framework to work on large data set with big number of nodes. And for these conditions, spark gives its best performance.

4. CAN YOU PREDICT WHICH WOULD BE BEST AT 100 NODE SCALE?

Spark, it gives consistent good performance in terms of throughput.

5. HOW ABOUT 1000 NODE SCALES?

Spark, it gives consistent good performance in terms of throughput.

6. COMPARE YOUR RESULTS WITH THOSE FROM THE SORT BENCHMARK, SPECIFICALLY THE WINNERS IN 2013-14 WHO USED HADOOP AND SPARK.

The results in cloud sort are much higher compared to my results. This is so because they had used many more nodes as well as instances with more cores and more memory.

7. WHAT CAN YOU LEARN FROM THE CLOUD SORT BENCHMARK

Sorting benchmarks ae used to check efficiency and performance of cloud. External sort is representative of many IO-intensive workloads. It's a holistic workload that exercises memory, CPU, OS, file-system, IO, network, and storage. It's simple and therefore easy to port and optimize on cutting-edge technologies. Now a days clouds are much cheaper and easily accessible. Clouds also provide storage options to store large amount of data, Google Clod Storage, S3. And they very scalable. So, clouds are not just used to perform data intensive work, but also used for storing data. And clod sort gives different benchmark for this, so its easy for organizations to select which clod platform to be used.

CONCLUSION

By checking all above result we can say that Spark gives best performance between Shared Memory, Hadoop and Spark.

Why Spark is Faster?
There are two main reasons:

1. One of the main limitations of Map Reduce is that it persists the full dataset to HDFS after running each job. Spark takes a more holistic view of a pipeline of operations. When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage. This is an innovation over Map Reduce that came from Microsoft's Dryad paper, and is not original to Spark.

2. The main innovation of Spark was to introduce an in-memory caching abstraction. This makes Spark ideal for workloads where multiple operations access the same input data. Users can instruct Spark to cache input data sets in memory, so they don't need to be read from disk for each operation.