PA2 Project Report

A20379025 Hsueh Yi Chen
A20385942 Yitong Huang

Shared Memory TeraSort

- The program is implemented to run terasort in big data size which is larger than the memory.
- First we use gensort program (download the source code from http://www.ordinal.com/gensort.html) to generate 128GB and 1TB data set. The data set generated text data with each data record having size of 100 bytes. Within each data record, the first 10 bytes are distinct text values, the rest are record counter and tabs. We should take the data format into account when implementing the shared memory terasort program.
- We split the program into three parts.
  1) Main function: it has main function to run the program. It chunked data into fixed size, we tried chunked data into both 1 MB and 2MB to test the performance. To achieve the peak performance, the function also created threads to call external sort and merge functions.
  2) Sort function: it sorted chunked data in place with merge sort.
  3) Merge function: it merged chunked files two by two with naive merge approach, and finally merge the sorted file into one output file.

System Configuration:

- Amazon machine image (AMI):
  Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-82f4dae7
  Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical
- With 128GB terasort, we used Amazon 1 x i3.large instance.
  i3.large (9 ECUs, 2 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 15.25 GiB memory, EBS only) 1 x 475 (SSD)
- With 1TB terasort, we used Amazon1 x i3.4xlarge instance.
  i3.4xlarge (53 ECUs, 16 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 122 GiB memory, EBS only)
- We mounted the EBS volume to boot and combine the disks into a RAID-0 to achieve the best possible performance.

Experiment 1:

- Experiment preparation: Before running 128GB data set, we tested on small dataset with 1GB text data in micro instance since 128GB experiment will take a long time. The result shows change the number of threads and number of chunked files will not lead to a significant difference in computing time performance. Even worse, in instance without enough cores, the more number of threads will cause a lot more time and will sometimes, cause output file out of order.
  The result of experiment preparation is shown in below table:

	Chunked size		
Thread number		1MB	2MB
	1	243sec	241sec
	2	218sec	291sec
	4	233sec	190sec
	8	220sec	459sec

Table1 1GB terasort with multiple thread number and chunked size

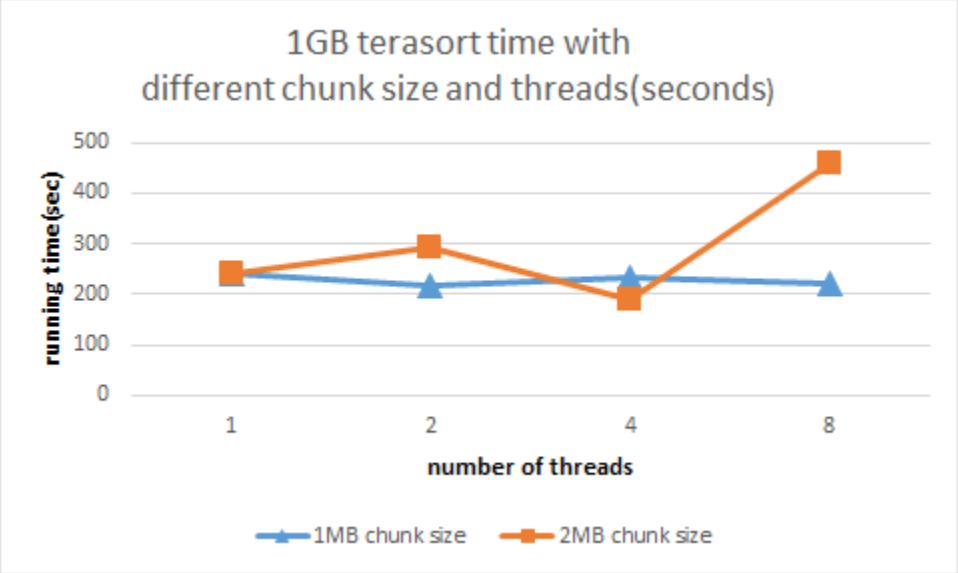


Figure1 experiment preparation with 1GB terasort

- With the data preparation, we decided to run 128GB data in with 1 thread since we only have 2 vCPU here, and we use 1MB data chunk size. The result is shown in table 2 below.

Experiment 2:

- According to the experiment preparation in experiment 1, we did not increase the chunked size too much for experiment 2. In this experiment, we used 2MB chunk size to see if any speed up with increased chunked size. Also we used 8 threads to run this experiment since we have 16 vCPU in this instance.

- The result is shown in table 2 below.

Performance Calculation for Experiment 1 and 2:

1. [1xi3.large 128GB]

- Compute Time (sec):
  23323 seconds
  This time is counted directly within the program.

- Data Read (GB):
  We used naive two-way merge in the implementation.
  Hence,
  data read(GB)=data size read per time x read times = data size read per time x log (number of blocks after chunking)
  Here the data has chunk size 1MB, so after chunking, the number of blocks in the first level is 128G/1MB=128000.
  data read(GB) = 128 X log128000 = 128 x 17 = 2176G.

- Data Write (GB) **:**
  data write(GB)=data size write per time x write times = data size write per time x log (number of blocks after chunking)
  So here the total data write size equals to the data read size. It is also 2176G.

- I/O Throughput (MB/sec) **:**
  Since total computing time is 23323 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  =2176 x 2 x 1000 /23323 = 186MB/sec

2. [1xi3.4xlarge 1TB]:

- Compute Time (sec)**:**
  71462 seconds.
  This time is counted directly within the program.

- Data Read (GB):
  We used naive two-way merge in the implementation.
  Hence,
  data read(GB)=data size read per time x read times = data size read per time x log (number of blocks after chunking)
  So here the data has chunk size 2MB, so after chunking, the number of blocks in the first level is 1TB/2MB=500,000,000.
  data read(GB) = 1000 X log500000000 = 1000 x 28.89 = 28890G.

- Data Write (GB):

data write(GB)=data size write per time x write times = data size write per time x log (number of blocks after chunking)
So here the total data write size equals to the data read size. It is also 28890G.

- I/O Throughput (MB/sec):
  Since total computing time is 71460 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  28890 x 2 x 1000 / 71462 = 808MB/sec

3. **Speedup (weak scale) = 8x * efficiency**=8 x 0.33=2.64X

4. **Efficiency (weak scale) =128GB compute time / 1TB compute time** = 23323/71462=33%

Conclusion:

- Since the speedup is much lower than ideal speedup. Shared memory terasort is low scalable.
- Since the two experiment uses weak scale, that means the data size has 8X scale with resource 8X scale accordingly. The ideal/theoretical speedup should be 8X.
  The reason the speedup of our experiments is lower than ideal one is that, we used naive two way merging in our programming. It causes the real data read and write size is large than 8X. Actually in our experiments the data read and write size in 1TB experiment is over 13 (28890/2176) times the one of 128GB experiment.
  Since we have done experiments in 1GB data and it shows number of threads and file chunk size have no significant in compute time, we can say that the large data read and write size is the main reason of low speedup.
- As per our experiment preparation based on 1GB data terasort shown before, multi-threads is not a good method to achieve better performance, especially when the number of virtual nodes/CPU are limited. Sometimes, multiple threads with small instance will lead to failure of the results, i.e. sorting out of order. My explanation is different threads cannot handle the output in order sometimes; for example, one thread process the sorting and merging faster than another slower thread, and the faster thread writes the result to disk file earlier than the slower thread. If the slower thread has a smaller record to write, it will lead the sorting out of order.
- As per our experiment preparation based on 1GB data terasort shown before, changing size of chunked data does not contribute a lot in performance improvement. It might because we did not change the data size to a large one due to the long experiment time. Theoretically, as long as chunked data size fit into memory, larger chunked data size will

decrease the computing time size it will decrease the read time and write time. We can improve it if we have more time in the future.

Hadoop Terasort

- First we use hadoop-mapreduce-examples-2.7.4.jar (hadoop sample program) to generate 128GB and 1TB file in HDFS directly. Then we use mapreduce program to sort each file.
- We split the program into three parts.
  1) Main function: it has main function to run the program. In the beginning, it created a job instance then start to set mapper, combiner and reducer function in this job. Finally, it implement each function to sorting the data. To achieve the peak performance, the program also call combiner, which grouped data in the map phase.
  2) Mapper function: it split each line of data in <key, value> pair, which key is first 10 characters, and value is remaining characters. Then it passed all <key, value> pair on combiner function.
  3) Reducer function: it took the grouped data from combiner as input and sorting all of the values associated with that key.

System Configuration:

- Amazon machine image (AMI):
  Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-82f4dae7
  Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical
- With 128GB terasort, we used Amazon ubuntu image 1 x i3.large instance.
  i3.large (9 ECUs, 2 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 15.25 GiB memory, EBS only) 1 x 475 (SSD)
- With 1TB terasort, we used Amazon ubuntu image 1 x i3.4xlarge instance.
  i3.4xlarge (53 ECUs, 16 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 122 GiB memory, EBS only)
- With 1TB terasort in cluster, we used Amazon ubuntu image 8 x i3.large instance.
  i3.large (9 ECUs, 2 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 15.25 GiB memory, EBS only) 1 x 475 (SSD)
- Environment setting: java-8-openjdk-amd64, hadoop-2.7.4
- We mounted the EBS volume to boot and combine the disks into a RAID-0 to achieve the best possible performance.

Experiment 1:

- Experiment preparation: Before running 128GB data set, we installed java environment and hadoop package. Then we start to set each hadoop configuration file hdfs-site.xml,

mapred-site.xml, core-site.xml and yarn-site.xml. Next we set environment variables in ./bashrc and hadoop-env.sh. Then we can format namenode and start hadoop framework.

- We started testing small dataset like 1GB file to check hadoop is mounted successfully. Then we tested 128GB using 1 mapper and 1 reducer , but it was fail due to lack of disk size. So we mounted 300GB EBS to accommodate datanode and namenode files and 400 GB EBS for temperate data which is generated from mapreduce program. The result is shown in Figure 2 to compare experiment 2 result.

Experiment 2:

- Experiment preparation: Different preparation with 128GB is hadoop configuration. Regarding hdfs-site.xml, we change block size from 64MB to 1GB. Regarding mapred-site.xml, we enlarge amount memory of mapper and reducer from 1024MB to 2GB.
- According the 1TB file size, we also need to enlarge EBS volume to accommodate temperate files and datanode/namenode files. To further explain, we put datanode/namenode files inside 2TB disk and temperate files inside 3TB disk.

Experiment 3:

- Experiment preparation: We needed to set up the basic hadoop environment, java and hadoop in 8 nodes first, so we used install script to run each node to make sure their configuration are the same. we also set up password-less login between one namenode and seven datanodes.
- Before running 1TB file size, we test 1 GB file to make sure mapreduce can work well in the cluster. Then we started generate 1TB file in HDFS, which can avoid lack of disk size problem in local nodes. However, it was fails to sort successfully when Yarn resource manager is out of resource to give reducer. Therefore, we enlarged RAM allocation setting in yarn-site.xml and mapred-site.xml to solve this issue.
- Also, we increase the number of mapper and reducer from 1 to 2, which can sort more efficiently. Replication number also changes from 1 to 3, which can achieve fault tolerance.

The result of experiment is shown in below table:

	file size		
node number		128GB	1TB
	1	20115 sec	67268 sec
	8	N/A	33268 sec

Table 2 128GB and 1TB terasort with different nodes

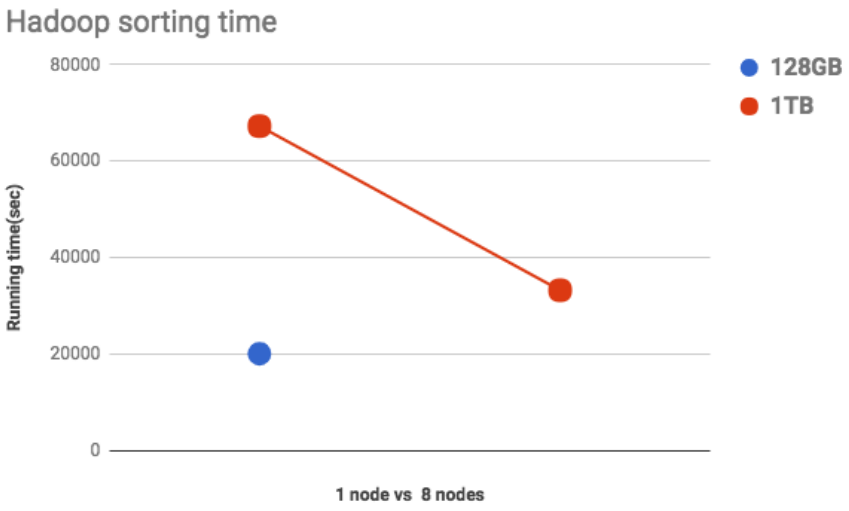


Figure 2 chart of 128GB and 1TB terasort with different nodes

Performance Calculation among Experiment 1,2 and 3:

1. [1xi3.large 128GB]

- Compute Time (sec):
  20115 seconds
  This time is counted directly within the program.

- Data Read (GB):
  61278 GB
  This size is counted directly within the result-File system Counters.
- Data Write (GB)：
  128 GB
  This size is counted directly within the result-File system Counters.

- I/O Throughput (MB/sec)：
  Since total computing time is 20115 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  =(61278+128)x 1000 /20115 = 3050 MB/sec

2. [1xi3.4xlarge 1TB]:

- Compute Time (sec)：
  67268 seconds.
  This time is counted directly within the program.

- Data Read (GB):
  3727284 GB
  This size is counted directly within the result-File system Counters.

- Data Write (GB):
  1000 GB
  This size is counted directly within the result-File system Counters.

- I/O Throughput (MB/sec):
  Since total computing time is 67268 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  (3727284+1000) x 1000 / 67268 = 55424 MB/sec

3. [8xi.large 1TB]:

- Compute Time (sec)**:**
  33268 seconds.
  This time is counted directly within the program.

- Data Read (GB):
  465 GB
  This size is counted directly within the result-File system Counters.

- Data Write (GB):
  1000 GB
  This size is counted directly within the result-File system Counters.

- I/O Throughput (MB/sec):
  Since total computing time is 33268 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  (465+1000) x 1000 / 33268 = 44 MB/sec

4. **Speedup (weak scale) = 8x * efficiency**
   =8 x (33268/67268) = 3.95X
5. **Efficiency (weak scale) =1TB 1 node compute time / 1TB 8 nodes compute time**
   = 33268/67268 = 49%

Spark Terasort

- First we use hadoop-mapreduce-examples-2.7.4.jar (hadoop sample program) to generate 128GB and 1TB file in HDFS directly. Then we use spark RDD progrm to sort each file.

- We split the program into four parts.
  1) SC.**textFile** function: Spark use textFile to store file in RDD format.
  2) Map function: Spark use **flatmap** to split each in line of data, then use **map** to convert each line to <key, value> pair, which key is first 10 characters, and value is remaining characters.
  3) Reduce function: Spark use **sortByKey** to group key for each <key, value> pair, than use **reduce** action to change each <key, value> pair to line string object.
  4) SC.**saveAsTextFile**: Finally Spark use **parallelize** to transform object into RDD format, then use **saveAsTextFile** to store in external file system.

System Configuration:

- Amazon machine image (AMI):
  Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-82f4dae7
  Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical
- With 128GB terasort, we used Amazon ubuntu image 1 x i3.large instance.
  i3.large (9 ECUs, 2 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 15.25 GiB memory, EBS only) 1 x 475 (SSD)
- With 1TB terasort, we used Amazon ubuntu image 1 x i3.4xlarge instance.
  i3.4xlarge (53 ECUs, 16 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4, 122 GiB memory, EBS only)
- Environment setting: java-8-openjdk-amd64, hadoop-2.7.4,spark-2.2.0-bin-hadoop2.7, python3
- We mounted the EBS volume to boot and combine the disks into a RAID-0 to achieve the best possible performance.

Experiment 1:

- Experiment preparation: We use previous hadoop experiment environment and then install spark and python package. Then we changed environment variable in ./bashrc. Now we ready to run Spark sorting.

Experiment 2:

- Experiment preparation: Different preparation with 128GB is spark configuration. Regarding spark-defaults.conf, we set spark.kryoserializer.buffer.max 2045MB in order to solve insufficient memory problem. we change block size from 64MB to 1GB. Also we increase executor memory and core, with 8g and 5 respectively, in order to get the good performance more efficiently.

  The result of experiment is shown in below table:

	file size		
node number		128GB	1TB
	1	10835 sec	36240 sec

Table 3 128GB and 1TB terasort

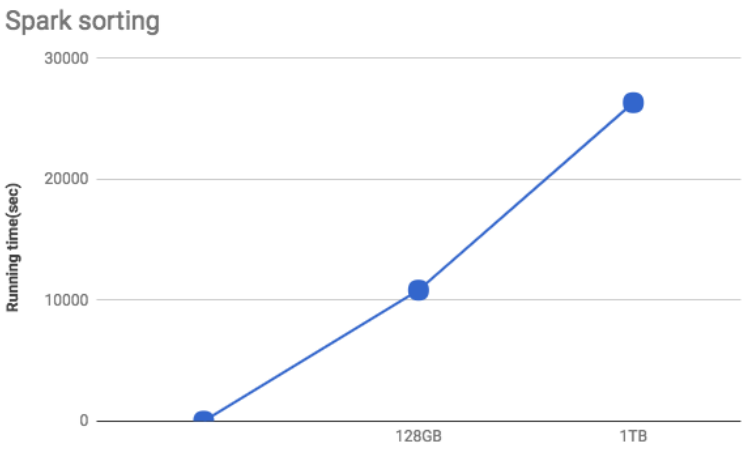


Figure 3 line chart of 128GB and 1TB terasort

Performance Calculation among Experiment 1 and 2:

1. [1xi3.large 128GB]

- Compute Time (sec):
  10835 seconds
  This time is counted directly within the program.

- Data Read (GB):
  128 GB
  This size is counted directly within the result-File system Counters.
- Data Write (GB)：
  128 GB
  This size is counted directly within the result-File system Counters.

- I/O Throughput (MB/sec)：
  Since total computing time is 10835 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  =(61278+128)x 1000 /20115 = 3050 MB/sec

2. [1xi3.4xlarge 1TB]:

- Compute Time (sec)：
  36240 seconds.

This time is counted directly within the program.

- Data Read (GB):
  1000 GB
  This size is counted directly within the result-File system Counters.

- Data Write (GB):
  1000 GB
  This size is counted directly within the result-File system Counters.

- I/O Throughput (MB/sec):
  Since total computing time is 36240 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  (3727284+1000) x 1000 / 67268 = 55424 MB/sec

3. [8xi.large 1TB]:

- Compute Time (sec)**:**
  33268 seconds.
  This time is counted directly within the program.

- Data Read (GB):
  465 GB
  This size is counted directly within the result-File system Counters.

- Data Write (GB):
  1000 GB
  This size is counted directly within the result-File system Counters.

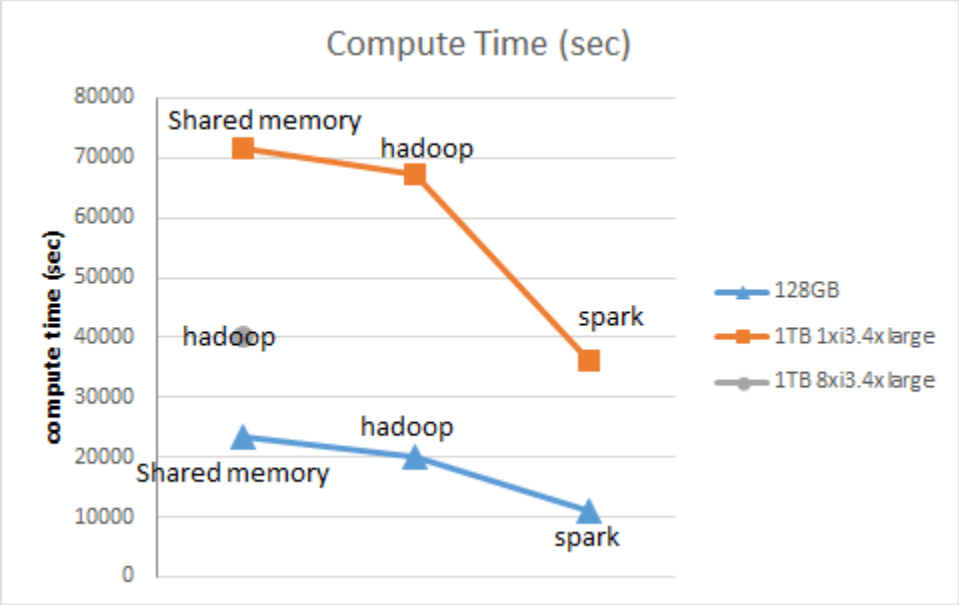
- I/O Throughput (MB/sec):
  Since total computing time is 33268 seconds,
  the I/O Throughput (MB/sec) = total data size (read and write)/total compute time
  (465+1000) x 1000 / 33268 = 44 MB/sec

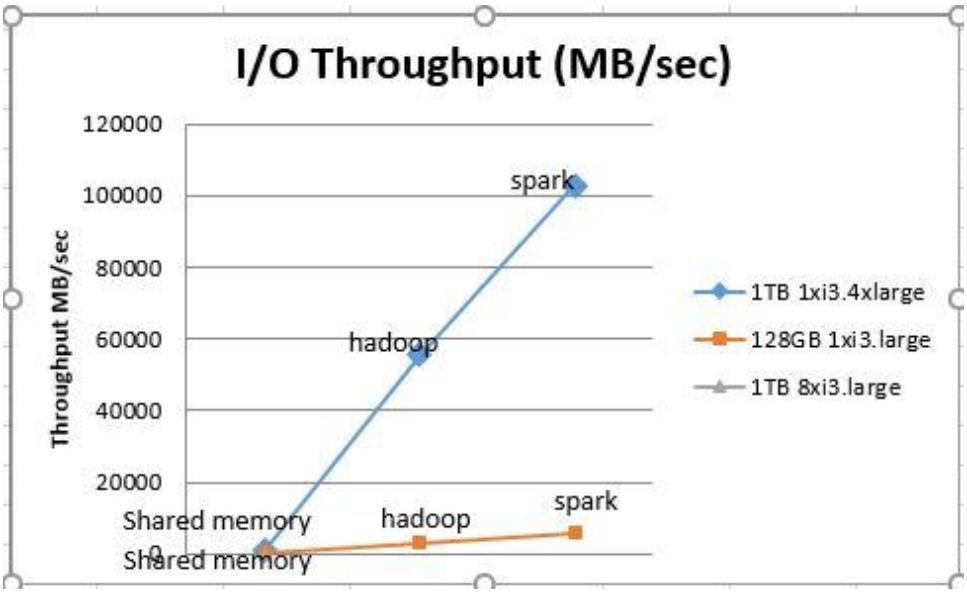
4. **Speedup (weak scale) = 8x * efficiency**
   =8 x (10835/33268) = 2.4X
5. **Efficiency (weak scale) =128GB compute time / 1TB compute time**
   = 10835/33268 = 30%

Experiment (instance/dataset)	Shared Memory TeraSort	Hadoop TeraSort	Spark TeraSort
Compute Time (sec) [1xi3.large 128GB]	23323	20114	10835
Data Read (GB) [1xi3.large 128GB]	2176	61278	61278
Data Write (GB) [1xi3.large 128GB]	2176	128	128
I/O Throughput (MB/sec) [1xi3.large 128GB]	186	3050	5667
Compute Time (sec) [1xi3.4xlarge 1TB]	71462	67268	36240
Data Read (GB) [1xi3.4xlarge 1TB]	28890	3727284	3727284
Data Write (GB) [1xi3.4xlarge 1TB]	28890	1000	1000
I/O Throughput (MB/sec) [1xi3.4xlarge 1TB]	808	55424	102877
Compute Time (sec) [8xi3.large 1TB]	N/A	40000	
Data Read (GB) [8xi3.large 1TB]	N/A	465	
Data Write (GB) [8xi3.large 1TB]	N/A	1000	
I/O Throughput (MB/sec) [8xi3.large 1TB]	N/A	44	
Speedup (weak scale)	2.64X	3.95	2.4
Efficiency (weak scale)	33%	49%	30%

Table 2 Performance evaluation of TeraSort

- Comparing computing time and throughput by figure





- Which seems to be best at 1 node scale?
  Spark performs best at 1 node compare to shared-memory and Hadoop compare to Throughput Comparison graph.
- Can you predict which would be best at 100 node scale?
  Regarding our experiment, Spark would be best for 100 node scale because it stores the intermediate data so that we can directly use those data when we want at any time but the only constraint is that spark consumes a huge amounts of memory.
- How about 1000 node scales?
  It depends on the system what we are using and how many resources we have. Because Spark takes large system resources. Though Spark is 100 times faster than Hadoop, it's not very friendly to intensive jobs. So still for intensive and large scale jobs, Hadoop would be the best choice for 1000 nodes.
- Compare your results with those from the Sort Benchmark, specifically the winners in 2013 and 2014 who used Hadoop and Spark.
  We can see 2013 Hadoop has throughput 1.42TB/min(23666666MB/sec), 2014 Apache Spark has throughput 4.27TB/min(71166666mb/sec), they are much faster than our shared memory performance. Because they have different processor and disk, and they use Hadoop and Spark, it will be more reasonable to compare with the Hadoop and Spark results below. But we can still see a significant advantage in Hadoop and Spark in terasort compare with shared memory.
  Cite: http://sortbenchmark.org/
- What you learn from the CloudSort benchmark.
  We can see cloud sort has great advantages in external sort. Because it is more accessible to normal people, people don't have to have access to national labs or some

institutions to do that. And the cloud platform like Amazon EC2 provides a reasonable and affordable price for us to do external sort on cloud. Moreover, with lots of previous work in this area, we can easily compare our results with some sorting benchmark and try to improve our own performance.