SORT ON HADOOP/SPARK

Abstract

Perform sorting of various datasets using linux sort, external merge sort, Hadoop & spark and compare their results

Mohit Aggarwal
Siddhant Jain

AIM:

The goal of this programming assignment is to enable you to gain experience programming with:

1. The Hadoop framework (http://hadoop.apache.org/)
2. The Spark framework (http://spark.apache.org/)

Linux & External Sort:

The internal memory sort was implemented as merge sort. Even though merge sort requires extra O(N) space, it always gives O(NlogN) time complexity. While quicksort doesn't need the extra memory, it can give O(N*N) time complexity in worst case and the implementation to make sure it only gives O(NlogN) in worst case requires implementation of further algorithms such as median of medians to get a good partition which increases the complexity of the code much further and was thus not implemented.

Thus, if the size of the file is less than half of the memory available, then the data will be sorted in the memory or it will be sorted using the external merge sort.

The system on which the experiment was done is an SSD and to make sure to get the maximum utilization from the disk, we used Java's Runnable class to use multi-threading while performing the division of the large files into multiple chunks.

The program requires user to provide the input file name, output file name and the number of threads. No. of threads if left for the person running to choose because various systems can have varying performance for the different number of threads and it's thus best if the user has the flexibility on his/her understanding.

Further, the caches was cleared to make sure the file is not in the buffer cache by using:

sudo sysctl -w vm.drop_caches=1

sudo sysctl -w vm.drop_caches=2

sudo sysctl -w vm.drop_caches=3

Also, by default Java doesn't use more than 8GB of memory while running. To utilize all the memory we passed the parameters from -Xmx8g to -Xmx32g for various configurations. While the naïve approach would be that increase in memory will result in increase in performance, this was not the case. This is due to the fact that when the memory size becomes very large, java ends up spending majority of it's time garbage collection once memory starts to get full. The most optimum value of memory can vary based on multiple factors like architecture, type of workload etc. for this use case, we get the best performance at about 8GB of ram while using OpenJDK 11. No previous version of JDK should not be used since won't only this result in poor performance but can also cause memory leak and crash.

Hadoop Sort:

In Hadoop, the process by which the intermediate output from mappers is transferred to the reducer is called Shuffling.

Shuffling is the process by which it transfers mappers intermediate output to the reducer. Reducer gets 1 or more keys and associated values on the basis of reducers. The intermediated key – value generated by mapper is sorted automatically by key. In Sort phase merging and sorting of map output takes place. Shuffling and Sorting in Hadoop occurs simultaneously.

Shuffling in MapReduce is the process of transferring data from the mappers to reducers is shuffling. It is also the process by which the system performs the sort. Then it transfers the map output to the reducer as input. This is the reason shuffle phase is necessary for the reducers. Otherwise, they would not have any input (or input from every mapper). Since shuffling can start even before the map phase has finished. So, this saves some time and completes the tasks in lesser time.

Sorting in MapReduce -MapReduce Framework automatically sort the keys generated by the mapper. Thus, before starting of reducer, all intermediate key-value pairs get sorted by key and not by value. It does not sort values passed to each reducer. They can be in any order.

Spark Sort:

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast queries against data of any size. Simply put, Spark is a fast and general engine for large-scale data processing.

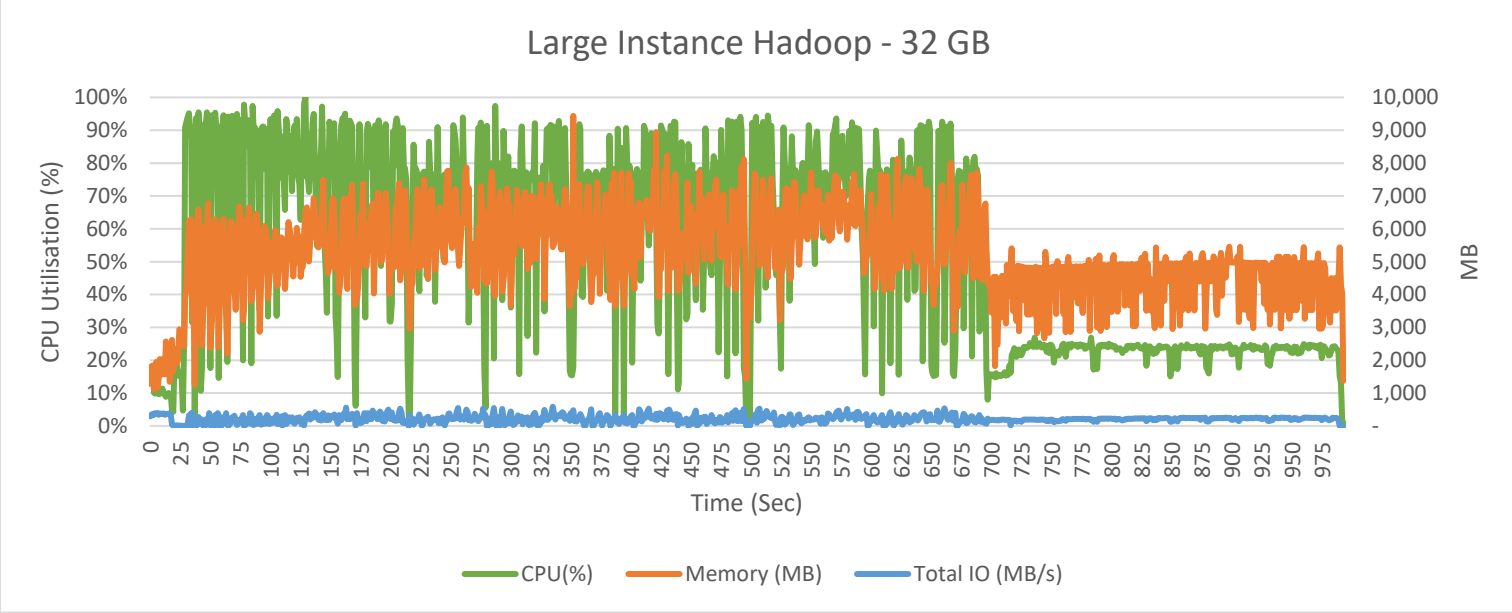
Much like MapReduce, Spark works to distribute data across a cluster, and process that data in parallel. The difference is, unlike MapReduce—which shuffles files around on disk—Spark works in memory, making it much faster at processing data than MapReduce.

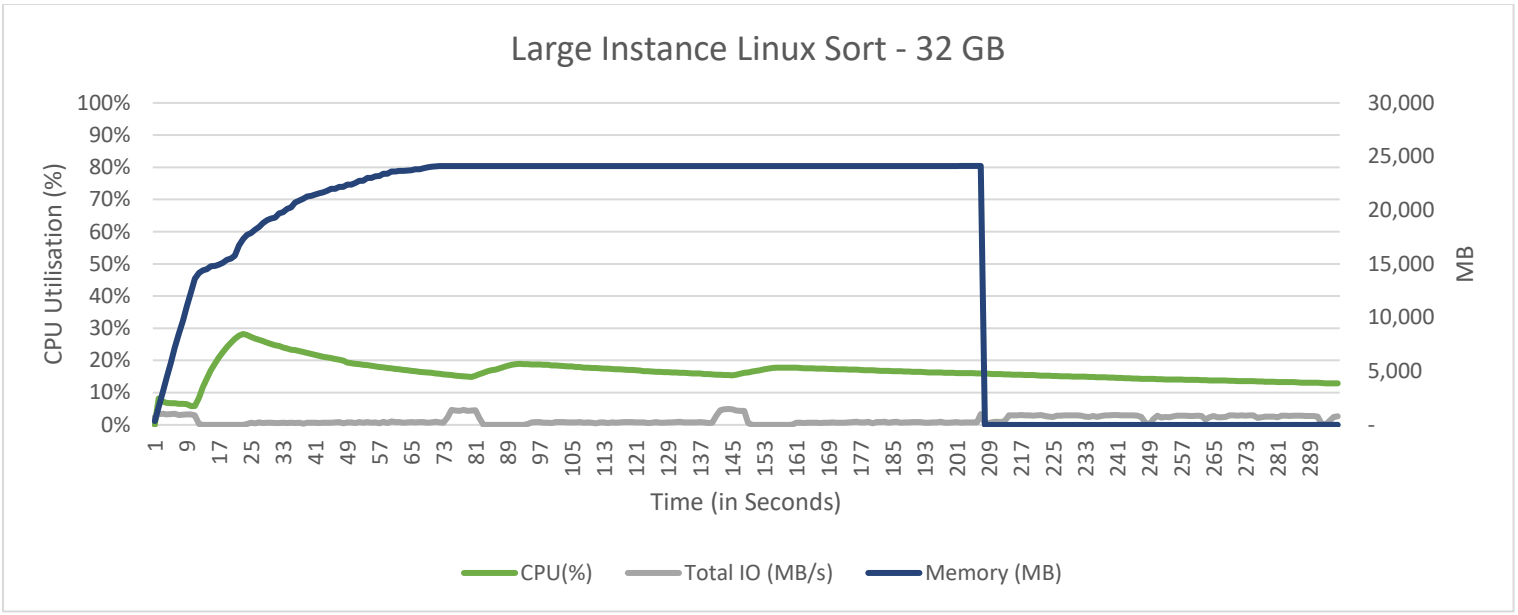
Spark can run on Apache Hadoop clusters, on its own cluster or on cloud-based platforms, and it can access diverse data sources such as data in Hadoop Distributed File System (HDFS) files, Apache Cassandra, Apache HBase or Amazon S3 cloudbased storage.

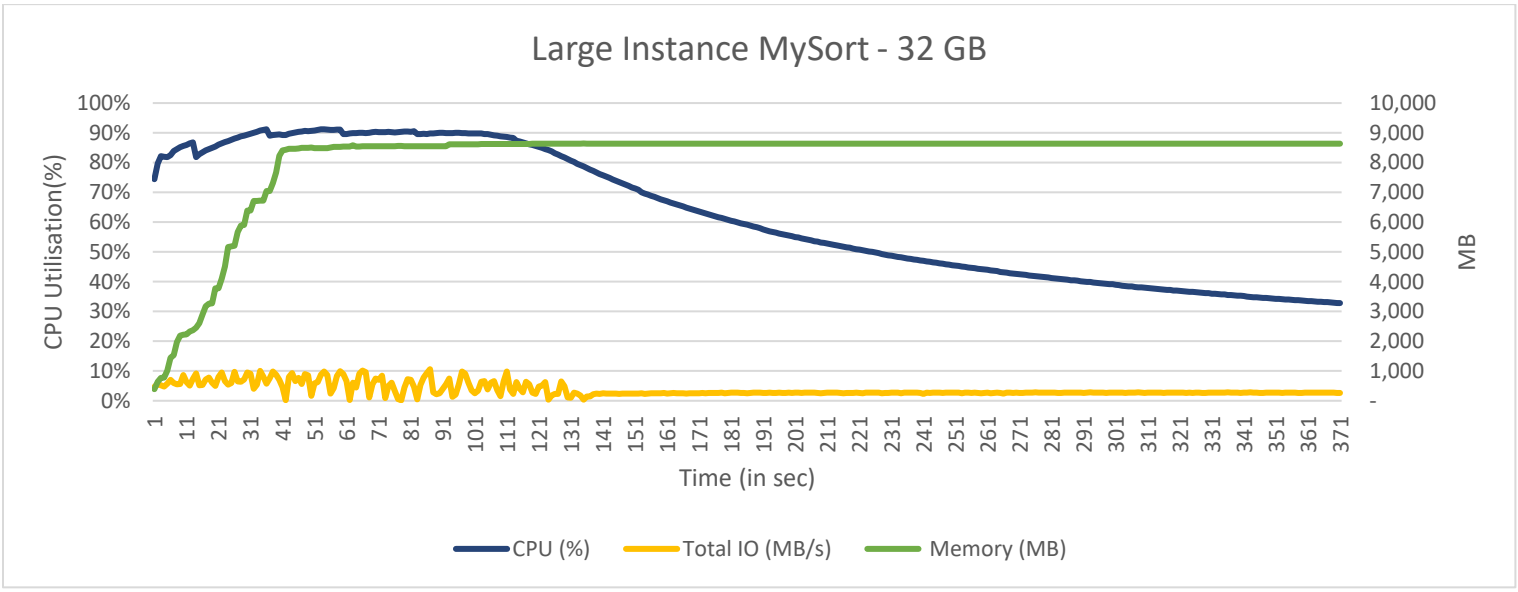
Results:

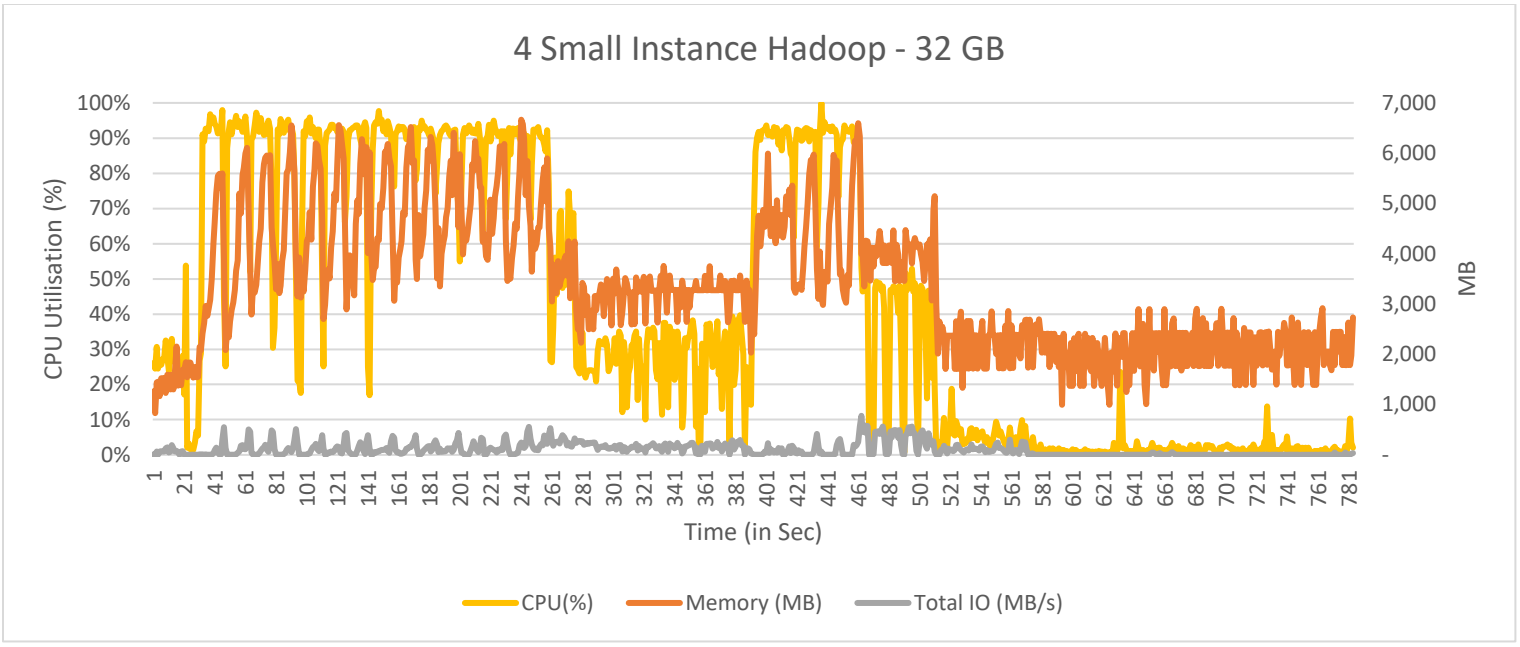
Experiment	Shared Memory	Linux	Hadoop Sort	Spark Sort
1 small.instance, 1GB dataset	25158	11469	55543	18564
1 small.instance, 4GB dataset	43856	59124	172476	167248
1 small.instance, 16GB dataset	165826	255010	670864	689578
1 large.instance, 1GB dataset	25000	7000	32622	35872
1 large.instance, 4GB dataset	62000	26000	70839	66070
1 large.instance, 16GB dataset	179000	154000	237198	237507
1 large.instance, 32GB dataset	383000	309000	984580	490000
4 small.instances, 1GB dataset	N/A	N/A	34114	35000
4 small.instances, 4GB dataset	N/A	N/A	77408	64000
4 small.instances, 16GB dataset	N/A	N/A	211991	217000
4 small.instances, 32GB dataset	N/A	N/A	960991	363000

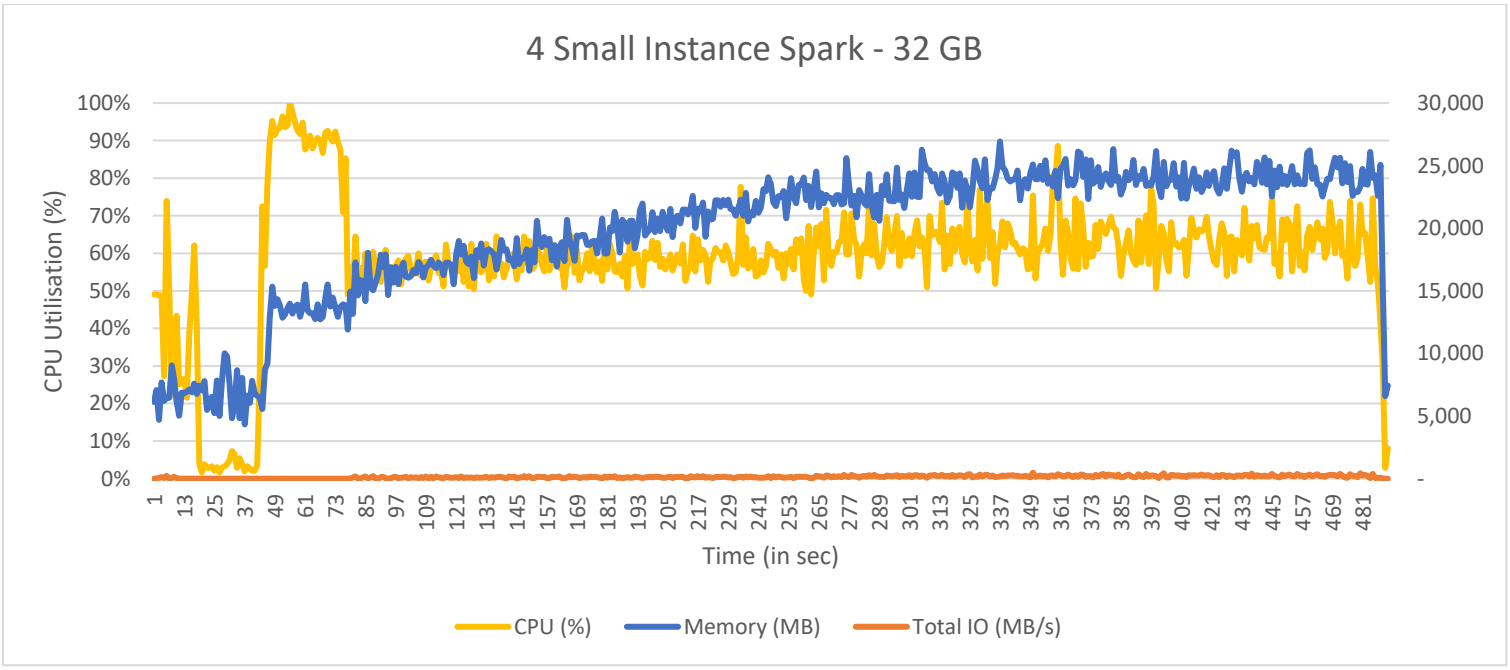
Times are in Miliseconds

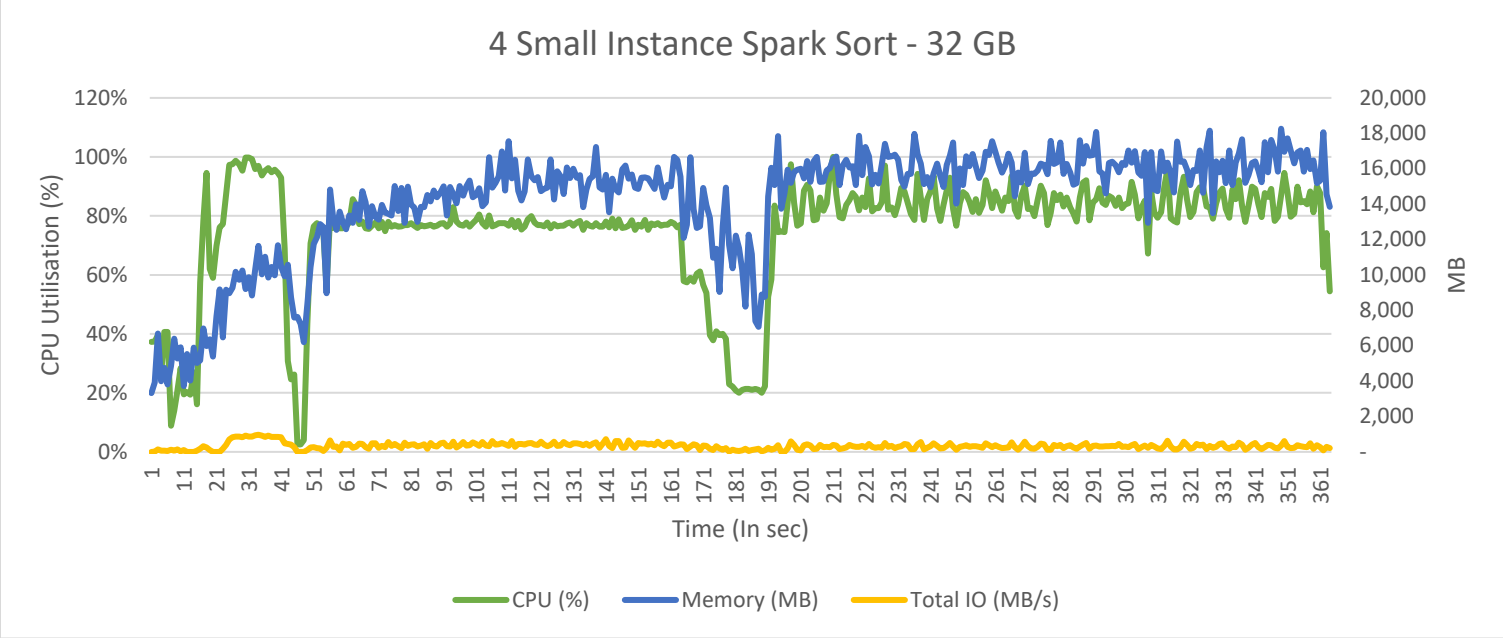



Large Instance Linux Sort - 32 GB
CPU Utilisation (%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
30,000
25,000
20,000
15,000
10,000
5,000
-
MB
1 9 17 25 33 41 49 57 65 73 81 89 97 105 113 121 129 137 145 153 161 169 177 185 193 201 209 217 225 233 241 249 257 265 273 281 289
Time (in Seconds)
CPU(%)
Total IO (MB/s)
Memory (MB)


Large Instance MySort - 32 GB
CPU Utilisation(%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
-
MB
1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311 321 331 341 351 361 371
Time (in sec)
CPU (%)
Total IO (MB/s)
Memory (MB)


4 Small Instance Hadoop - 32 GB
CPU Utilisation (%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
7,000
6,000
5,000
4,000
3,000
2,000
1,000
-
MB
1 21 41 61 81 101 121 141 161 181 201 221 241 261 281 301 321 341 361 381 401 421 441 461 481 501 521 541 561 581 601 621 641 661 681 701 721 741 761 781
Time (in Sec)
CPU(%)
Memory (MB)
Total IO (MB/s)


4 Small Instance Spark - 32 GB
CPU Utilisation (%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
30,000
25,000
20,000
15,000
10,000
5,000
-
1 13 25 37 49 61 73 85 97 109 121 133 145 157 169 181 193 205 217 229 241 253 265 277 289 301 313 325 337 349 361 373 385 397 409 421 433 445 457 469 481
Time (in sec)
CPU (%)
Memory (MB)
Total IO (MB/s)



Conclusion

Q1. What conclusions can you draw?
Ans1. Linux and External sort gives very good performance and this could be attributed to the fact that we don't have any similar overhead in them unlike Hadoop and Spark. Also, multiple cores can be utilized to provide parallelization similar to Hadoop and Spark. But there is only a limit to how much we can sort using Linux and MySort because they require one machine while Hadoop and Spark can run on multiple machines and hence provide efficient scaling.
Also, Spark Sort beats Hadoop Sort in every test and this can be expected since spark has lesser disk I/Os than Hadoop since it utilizes very high amount of memory

Q2. Which seems to be best at 1 node scale (1 large.instance)? Is there a difference between 1 small.instance and 1 large.instance?
Ans2. At 1 node scale (1 large instance), Linux Sort seems to be the best. This can be due to the fact that both Hadoop and Sort have considerable overhead while MySorts performance although similar but still not as good as Linux Sort.
Yes, there is a difference between 1 small and 1 large instance since we have resources in the large instance and also small instance is limited to the amount of data we can sort, for ex, we can't sort a 32 GB dataset in it. Due to this sort timings are smaller in large instance but the various sorts performance is ranked similar to the small instance within large instance.

Q3. How about 4 nodes (4 small.instance)? What speedup do you achieve with strong scaling between 1 to 4 nodes?
Ans3. At 4 nodes(4 small instances), Spark Sort outperforms the Hadoop Sort and this is to be expected since Spark utilizes a lot more memory and lesser disk I/Os. This can be seen in the graph as well where memory utilization of Spark Sort can reach upto 24GB at times.
We are achieving a speedup of 2.6 for the 4 GB dataset and 3.2 for the 16GB dataset. We don't have any for 32 GB since we cannot run it on one node. For 1 GB, we actually see a decrease in performance. This is due to the fact that when using multiple nodes, we need to split the data and share it across the cluster which includes a certain overhead and since 1 GB is a really small dataset, this overhead outweighs the gain from parallelization.

Q4. What speedup do you achieve with weak scaling between 1 to 4 nodes? How many small.instances do you need with Hadoop to achieve the same level of performance as your shared memory sort?
Ans4. In case of 1 small instance, we are achieving a speedup of 2.3 by weak scaling. In case of 1 large instance, we don't see any speedup due to weak scaling. For the 4 nodes, we see the maximum speedup of 3.1 due to weak scaling.
We need about 8 small instances with Hadoop to achieve the same level of performance as shared memory sort.

Q5. How about how many small.instances do you need with Spark to achieve the same level of performance as you did with your shared memory sort?
Ans5. We need approximately 1.21 times in spark to achieve the same performance as memory sort

Q6. Can you draw any conclusions on the performance of the bare-metal instance performance from HW5 compared to the performance of your sort on a large instance through virtualization?
Ans6. Bare-metal performance is better than the virtualization as the time taken to sort same dataset has increased for both linux and external sort. This is due to the fact that we now have machines with lesser power and an extra middle layer overhead which further deteriorates the performance

Q7. Can you predict which would be best if you had 100 small.instances? How about 1000?
Ans7. Having 100 small instances gives the power of a very high parallelization and thus can be used to effectively sort very large datasets with lower costs.
Having 1000 instances can give higher performance power but with such large number of nodes, there are multiple issues like scheduling or coordinating between the nodes. This can give rise to new problems and without efficient management system can go in a deadlock or give weird results. Also, having such large number of instances provide redundancy since one node going down doesn't affect much of the systems performance although the probability of having a failure increases as the number of nodes increases.

Q8. Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark. Also, what can you learn from the CloudSort benchmark, a report can be found at (http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf).
Ans8. The winners of both 2013 and 2014 have an impressive performance considering the fact they are both sorting at a rate over 1.4 & 4.3 TB/sec. The stark difference between these two comes from the fact that the 2013 winner used Hadoop while 2014 one used Spark. As the datasets size increase, Spark will give better performance given that it has sufficient memory and all the parameters are set.
Same phenomenon can be noticed in our results where initially for small datasets Hadoop and Spark are more or less same but as the size increases so does the performance gap between them. We can also see in the graph that the Spark uses anywhere between 2 -3x memory of Hadoop and thus better performance due to low disk I/Os.

The paper proposes CloudSort as a new method for external sort benchmark CloudSort benchmarks external sort from perspective of Total Cost of Ownership(TCO). The authors argue that the current benchmark methods do not consider the management and maintenance cost of data centers. Since Cloud storage costs already considers the cost of management and maintenance therefore, they are ideal for this and hence give a full TCO.

They propose to run external sort on public clouds. The reason they use external sort as the sorting method to benchmark is because of it is representative of many IO Intensive workloads. It covers all

aspect such as memory, CPU, OS, file-system, IO network and storage while being simple and easy to port to other technologies.

Having the benchmarks on public clouds will give total perspective. They believe that with the current trend, enterprises will move on to public cloud. This will also assess efficacy of cloud-environments for IO intensive workload CloudSort aims to enable the transition. The reasons for their proposal are as follows:

1. Accessibility: Not everyone has access to top tier data labs and centers. Only people that have access to these kinds of national labs, companies or academic institution. CloudSort will level this playing field as everyone will have ability to access to public cloud services such as AWS, Azure etc.
2. Affordability: They state that the cost required to run 3 sort tests would cost only around $2000 hence it will economical enough for participants.
3. Auditability: It will be easier for others to verify the results as the participants can offer their sort software as virtual machine images.