Homework6 Report

Team-25

Hsuan-An Weng Lin (A20450355)
Kevin Tchouate Mouofo (A20454613)
Ji Tin Justin Li (A20423037)

Contribution:

Runtime environment setting: Hsuan-An Weng Lin
HDFS configuration: Ji Tin Justin Li, Hsuan-An Weng Lin
Hadoop Sort: Ji Tin Justin Li
Spark Sort: Hsuan-An Weng Lin
Performance monitoring: Hsuan-An Weng Lin
Performance graph plotting: Hsuan-An Weng Lin
Readme file: Ji Tin Justin Li, Hsuan-An Weng Lin
Report: Hsuan-An Weng Lin, Ji Tin Justin Li, Kevin Tchouate Mouofo

1. Problem description

This assignment was meant to assess the difference in performance between our sorting algorithms (in memory and external), linux sort, hadoop sort and spark sort. To do so, we have sorted some files of different size (1G, 4G, 16G and 32G) on different node: large instance (16-cores, 32GB ram, 200gb disk), 1 small instance (4-cores, 8GB ram, 50gb disk), and 4 small instances.

2. Methodology

To properly reach our goal, we have decided to go configuration by configuration. First, we have created our bare metal machine on Chameleon and have deployed Ubuntu 18.04 using “compute-skylake”. After that, we have created the environment of the large instance and have run all the sorting on that node. We recorded the results and monitored each sort. When we were done with the first instance, we deleted it and went on to the next configuration (1 small instance). We have followed the same process until the 3 configurations were done.

3. Runtime environment settings

To set up the environment of our experiments, we have installed a small NFS on our bare metal machine and mount it on all of our VMs. We stored the Hadoop and Spark binary files on the NFS so that we don't need to reinstall them again when we move on to next configurations. For the setting of hadoop, we turn off the replication option to reduce the disk usage of our experiments. Moreover, we also reserved some disk space on our data nodes by setting dfs.datanode.du.reserved configuration because we need to export the sorted file from the hdfs to validate the result.

4. Results and interpretation

<u>NB</u>: The hadoop generated intermediate files seems to be larger than anticipated. Because of disk constraint, we were unable to run the 64GB sort, we did 32GB instead.

Table 1: Performance evaluation of sort (report time to sort in milliseconds)

Experiment	Shared Memory	Linux	Hadoop Sort	Spark Sort
1 small.instance, 1GB dataset	1m12.754s	11.958s	54.603s	1m40.434s
1 small.instance, 4GB dataset	6m52.998s	1m7.475s	3m0.217s	4m17.068s
1 small.instance, 8GB dataset	14m38.137s	2m22.022s	6m3.144s	7m51.911s
1 large.instance, 1GB dataset	1m20.718s	11.489s	44.893s	1m36.874s
1 large.instance, 4GB dataset	7m1.747s	49.589s	2m30.069s	4m3.208s
1 large.instance, 16GB dataset	40m28.514s	4m6.934s	9m18.679s	15m30.850s
1 large.instance, 32GB dataset	80m34.560s	9m4.928s	18m15.583s	24m28.800s
4 small.instance, 1GB dataset	N/A	N/A	39.648s	1m8.765s
4 small.instance, 4GB dataset	N/A	N/A	1m58.229s	3m4.979s

4 small.instance, 16GB dataset	N/A	N/A	7m50.006s	12m34.803s
4 small.instance, 32GB dataset	N/A	N/A	N/A	N/A

Comparing the performance of 1 small.instance to 1 large.instance:

- ➔ For our *shared memory sort*, we can see the times required are similar (<10% difference) for small and large instances. This makes sense since the version of mySort we are running is only single threaded*, having more cores in the virtual machines does not (in theory) affect the program performance in any way.

  * Memory leakage bugs were found in our multithreaded mySort implementation when monitoring resource usage. To maintain robustness and repeatability of our experiments, older version of mySort is used.
- ➔ For *linux sort, hadoop sort and spark sort*, the large instance performs better than the small instance. These three sort programs all are able to utilize the available cores in the large instance, and thus performs better than the small instance.

Scalability analysis of 1 small.instance to 4 small.instances:

- ➔ With the experiment data collected, we can analyze the scalability of the programs from both weak scaling perspective and strong scaling perspective. We can see that from both perspectives, both hadoop sort and spark sort scales, but not perfectly.
  - ➔ Weak scaling: Comparing 1 small.instance(4GB) with 4 small.instances(16GB)
    - ◆ We can see for 1 node hadoop sort, it takes 3m0.217s. If it is completely unscalable, 4 node(16GB) would take 3m0.217s*4 ~= 12m. This theoretical value is larger than what we got in our practical experiment of 7m50.006s. Similar argument can be made for spark sort.
  - ➔ Strong scaling: Comparing 1 small.instance(4GB) with 4 small.instance(4GB)
    - ◆ For hadoop sort, the time reduction is easily observed: 3m0.217s for 1 node 4GB, compared with 1m58.229s for 4 node 4GB.
- ➔ For hadoop sort:
    - ◆ Strong scaling speedup = 3m0.217/1m58.229 = 180.217/118.229 = 1.524
    - ◆ Weak scaling speedup = 3m0.217*4/7m50.006s = 180.217*4/470.006 = 1.534
- ➔ For spark sort
    - ◆ Strong scaling speedup = 4m17.068s/3m4.979s = 257.068/184.979 = 1.390
    - ◆ Weak scaling speedup = 4m17.068*4/12m34.803s = 257.068*4/754.803 = 1.362

- ➔ If we look at the performance we had on HW5 compared to what we have on our large instance, we can see that we performed better on the bare-metal machine. This performance loss is probably due to the virtualization layer.

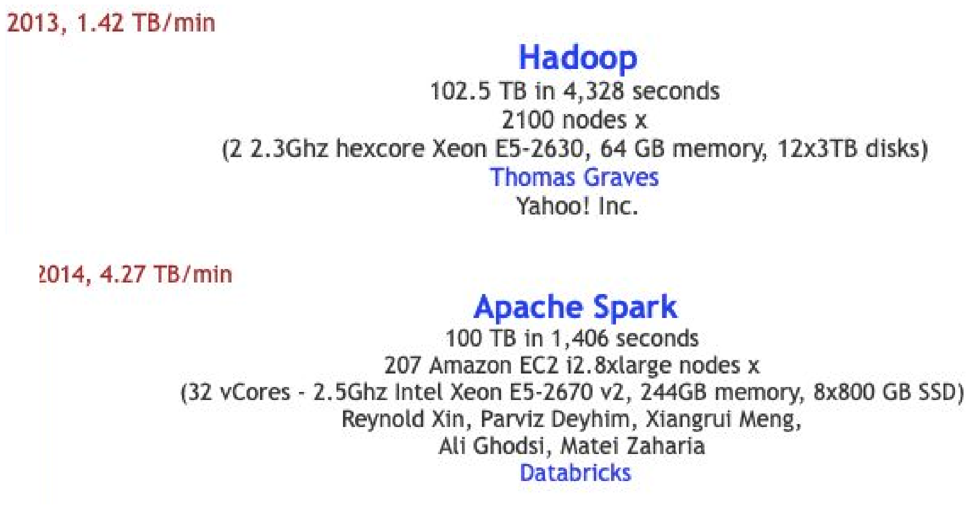
	Shared Memory HW5	**Shared Memory HW6 large**

		instance
1GB dataset	56.683s	1m20.718s
4GB dataset	4m74s	7m1.747s
16GB dataset	27m34s	40m28.514s

➔ According to our results, we believe if we have 100 small instances, hadoop will still have a better performance than spark. Even if we have 1000 small instances. However, we also believe that it is possible that we didn't utilize our Spark Sort program pretty well. According to the records on http://sortbenchmark.org, Spark can actually achieve very high performance when doing sorting. Therefore, we think if both of our Spark Sort and Hadoop Sort program is well utilized, the Spark Sort might have chance to have better performance within 100 or 1000 instances.

➔ As for scalability for the data size, spark sort produces better performance as we scale up the workload size. And for mySort, linux sort and hadoop sort, although not as well as spark, the performance scales decently i.e. the time required to sort scales (almost) proportionally to the datasize.

➔ Let's take a look at the sort benchmarks winners of 2013 and 2014 who used hadoop and spark:



2013, 1.42 TB/min

Hadoop
102.5 TB in 4,328 seconds
2100 nodes x
(2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks)
Thomas Graves
Yahoo! Inc.

2014, 4.27 TB/min

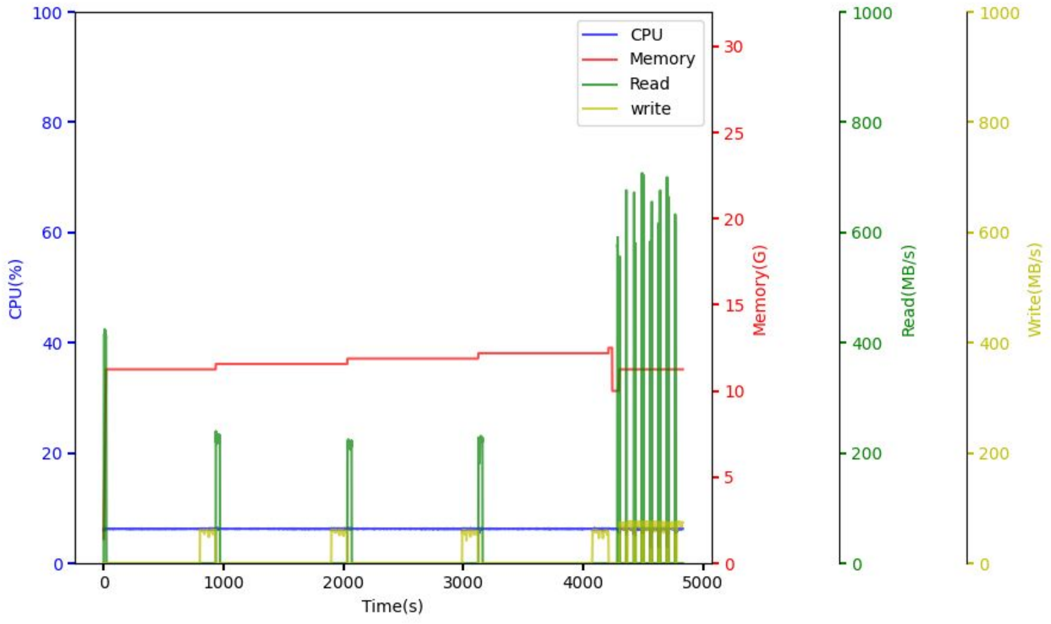
Apache Spark
100 TB in 1,406 seconds
207 Amazon EC2 i2.8xlarge nodes x
(32 vCores - 2.5Ghz Intel Xeon E5-2670 v2, 244GB memory, 8x800 GB SSD)
Reynold Xin, Parviz Deyhim, Xiangrui Meng,
Ali Ghodsi, Matei Zaharia
Databricks

With 4 small instances x (4-cores, 8GB ram, 50gb disk) we have sorted 16GB in 470s with hadoop, and 754s with spark. So we have a 0.002 TB/min with hadoop and 0.0013TB/min with spark. We can see that our hadoop sort and spark sort is obviously not well utilized. Although the winners of sort benchmarks use more nodes to do the sorting, we still have a lot of room to improve our programs.

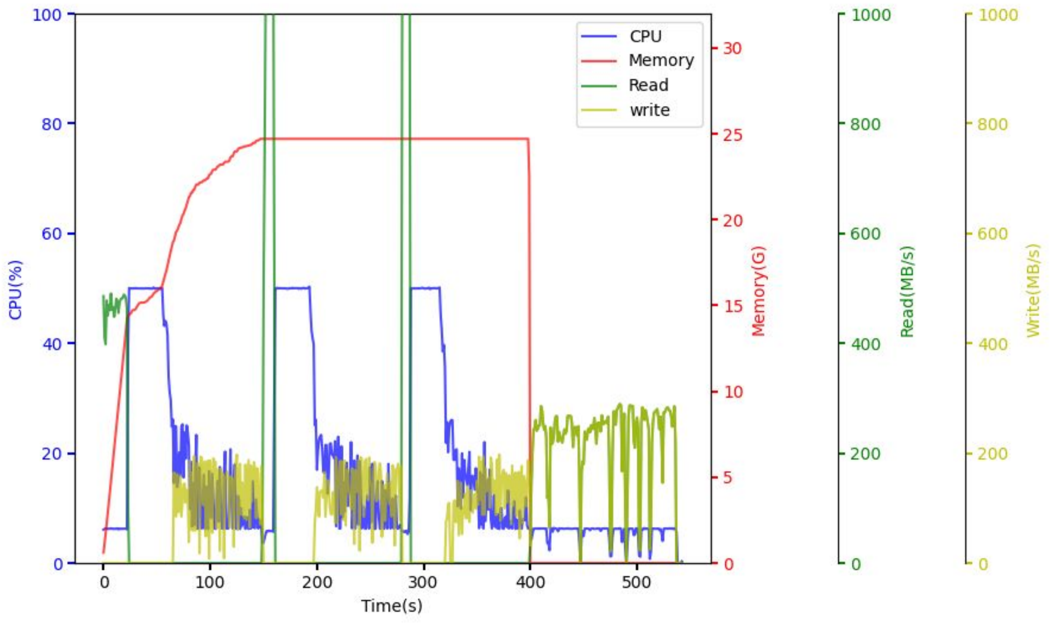
➔ The cloudSort benchmark is a benchmark that measures the efficiency of external sort from a total-cost of ownership perspective. It is a solution that would provide support for IO-intensive workloads. Therefore, if we are going to dig into the sorting area, this can give us an idea to benchmark our works properly.

5. Resource usage graphs and analysis

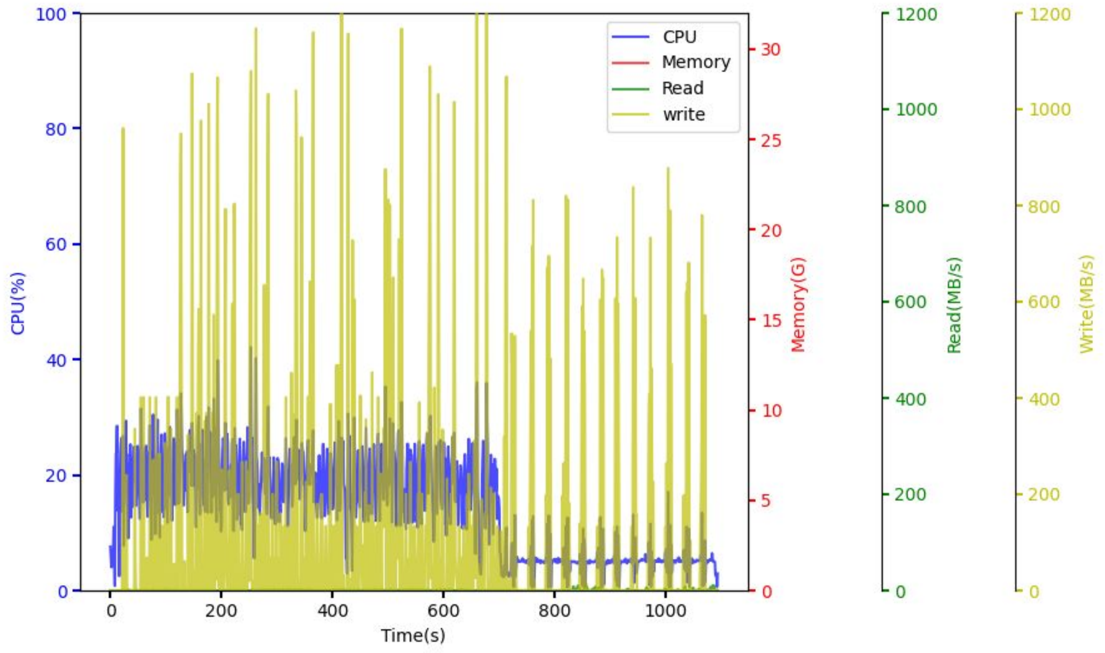
- Large.instance mySort 32G



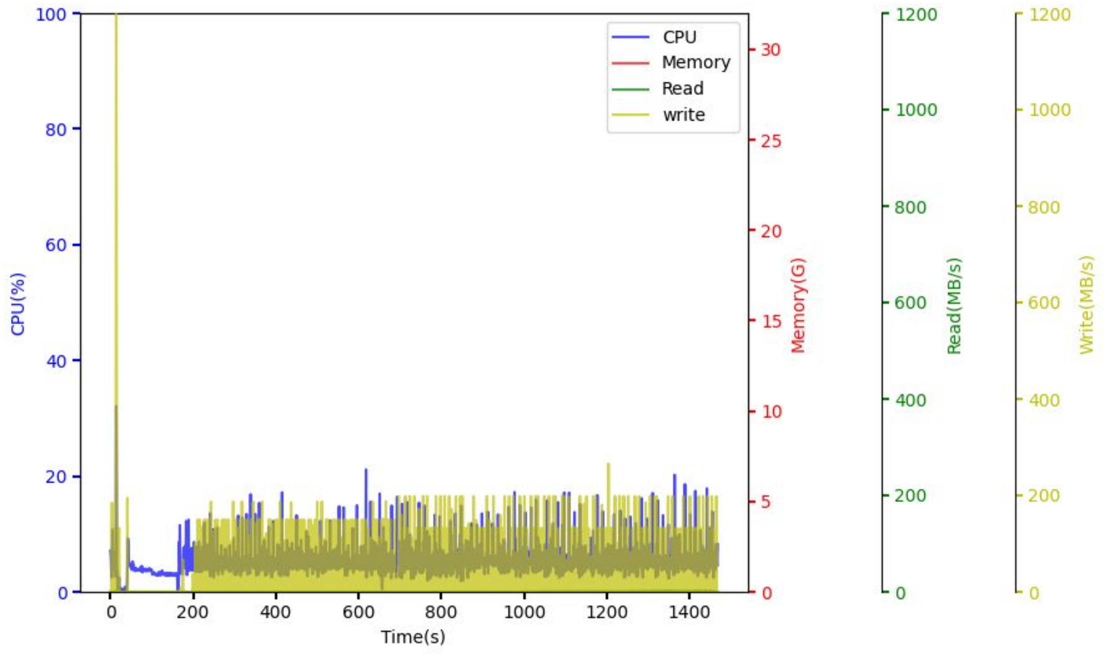
- Large.instance LinuxSort 32G



- Large.instance HadoopSort 32G



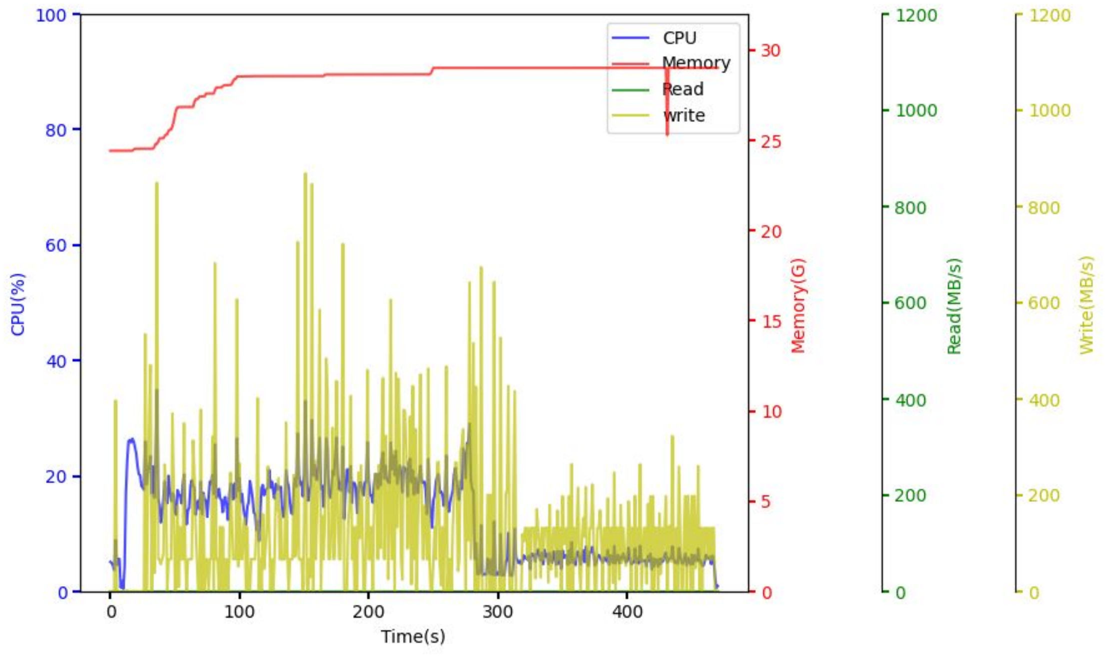
- Large.instance SparkSort 32G



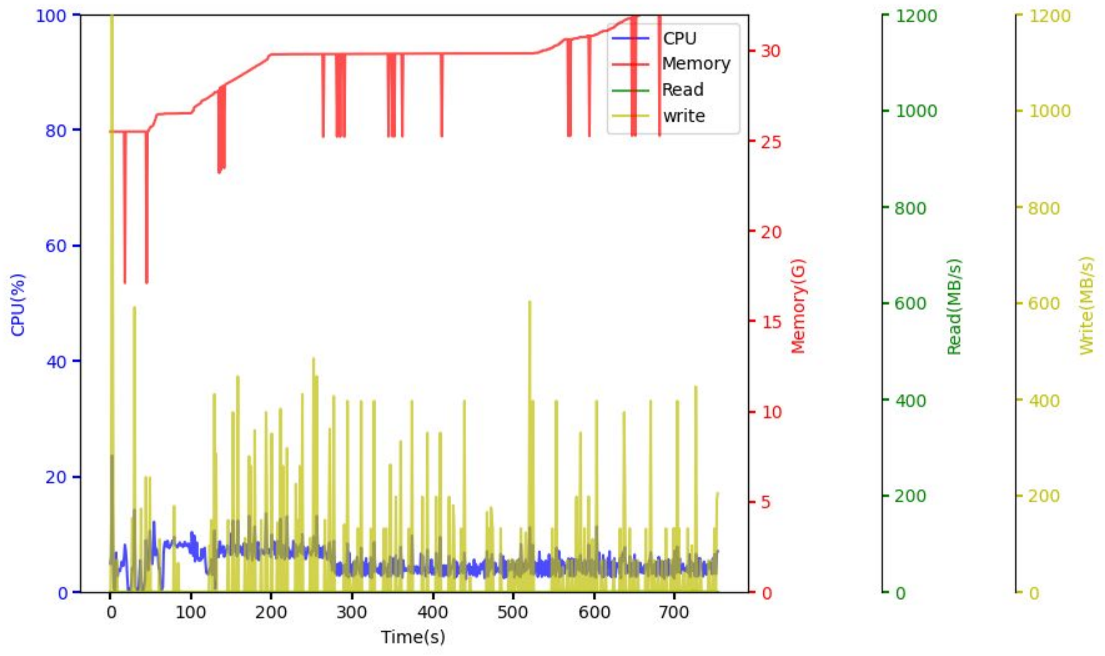
On the large instance, mySort presents 4 Peaks of read and write which are the internal sorting of each chunk, and the repetition of read and write in the end are the external sorting which we use k-merge technique to merge those sorted files. Compared to the linuxSort graph, the linuxSort graph seems to have a similar pattern with our mySort graph: internal sort first then external sort of merging files. However, linuxSort is obviously more efficient on the CPU and memory usage since it uses multithreading techniques to improve the overall performance which is not included in our mySort program. Moreover, our mySort program can't utilize the memory usage dynamically. We need to specify the memory size when we use our mySort and for all the experiments, we only assigned 8GB. This can explain why the memory usage on mySort graph is always close to 8GB.

For the Hadoop Sort and Spark Sort, the reason why it doesn't show any memory usage in Hadoop Sort and Spark Sort is because we monitor the performance of these two sorting methods by monitoring the whole VM on the host machine. In this case, the memory usage will be a fixed size that we assigned to the VMs. To solve this problem, we have tried to monitor the memory usage inside the VM by htop manually and it turns out that both Hadoop sort and Spark sort only use approximately 1-3GB memory. By looking into the graph of hadoop sort, we can see that there are two parts in the sorting process which can be divided by approximately 700sec. The first part(0-700sec) is the part that does mapping and the second part(700-end) is the part that does reduction. On spark sort graph, we observe that the performance is not as good as Hadoop Sort even though we have used RDD to do the Spark Sort. We believe that Spark should have better performance. Maybe we are not using the libraries properly. However, due to the time constraint, we didn't successfully utilize our Spark Sort program.

- 4*small.instance HadoopSort 16G



- 4*small.instance SparkSort 16G



For 4 small instances, due to the fact that the intermediate files generated by hadoop will fill the datanode and cause the VM crash, we decided to reduce the largest workload size to 16GB. Although we didn’t run the 32GB sorting on 4 small instances, we can still figure out something from the results of 16GB. To plot the graph of 4 small instances, we monitor the VMs on the host(bare metal machine) and then aggregate for outputs from those VMs. As we can see in the graph, the pattern of hadoop sort seems pretty similar to single instance - first mapping then reduction. And the IO performance doesn't seem to have improved much in multiple instances but still sort the dataset in less time. For Spark Sort, the IO performance is higher than the single instance which can explain why the efficiency of sorting a 16GB dataset on 4 instances is much higher than sorting it on only a single instance.