Cloud Computing
HW 6

- Problem
  Implementing Shared Memory Sort, Linux Sort, Hadoop Sort and Spark Sort on different configurations – 1 small instance, 4 small instance and 1 large instance using 4 different datasets – 1GB, 4GB, 16 GB, 32GB.

- Methodology
  - Shared Memory – Using parallel merge sort with in-memory and external sort for 1 GB & 4GB data and 16 GB and 32 GB data respectively.
  - Linux Sort – using sort command
  - Hadoop Sort – using Java and MapReduce Framework
  - Spark Sort – using Java and Spark Framework

- Runtime Environment Settings
  - Compute node: Skylake
    - No. of CPUs: 2
    - No. of Threads: 48
    - RAM size: 192 GiB
    - Disk: 240
  - 1 small instance
    - Namenode Configuration
      - 2cores
      - 4g ram
      - 15gb disk
    - Datanode Congiuration
      - 4 cores
      - 8gb ram
      - 50gb disk
  - 4 small instance
    - Namenode Configuration
      - 2 cores
      - 4g ram
      - 15gb disk
    - Datanode Congiuration x4
      - 4 cores
      - 8gb ram
      - 40gb disk
  - 1 large instance
    - 16 cores
    - 32gb ram
    - 200gb disk
  - For MySort: C++
  - For Hadoop and Spark: Java

➢ Performance Evaluation of Sort

➢ **Experiment**	**Shared Memory**	**Linux**	**Hadoop Sort**	**Spark Sort**
1 small.instance, 1GB *dataset*	98454.2	43311	58806	112156
1 small.instance, 4GB *dataset*	409258	212214	173355	287741
1 small.instance, 16GB *dataset*	2739739	995408	753951	998256
1 large.instance, 1GB *dataset*	92114.1	43727	42845	116803
1 large.instance, 4GB *dataset*	383898	196097	114728	288414
1 large.instance, 16GB *dataset*	2138510	981824	542916	1127413
1 large.instance, 32GB *dataset*	8303775	1968613	1105000	2124002
4 small.instances, 1GB *dataset*	N/A	N/A	41346	63000
4 small.instances, 4GB *dataset*	N/A	N/A	108498	163656
4 small.instances, 16GB *dataset*	N/A	N/A	742601	362990
4 small.instances, 32GB *dataset*	N/A	N/A	4042243	1215991

➢ Threads used for MySort and Linux Sort

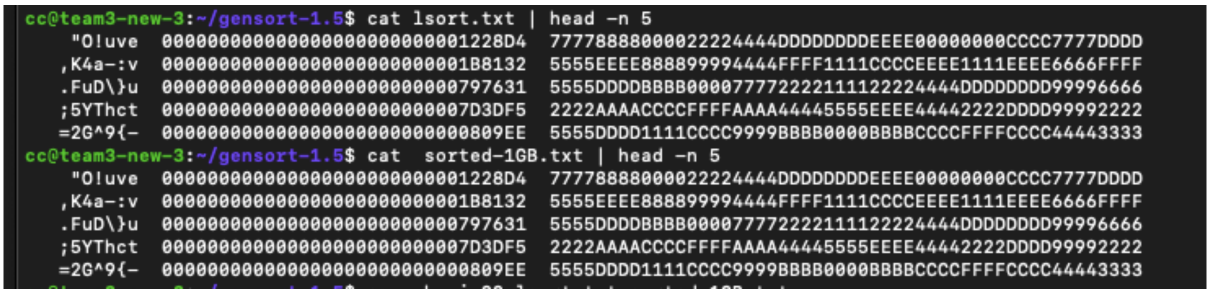
Experiment	**MySort**	**Linux Sort**
1 small.instance, 1GB *dataset*	24	24
1 small.instance, 4GB *dataset*	24	24
1 small.instance, 16GB *dataset*	48	48
1 large.instance, 1GB *dataset*	24	24
1 large.instance, 4GB *dataset*	24	24
1 large.instance, 16GB *dataset*	48	48
1 large.instance, 32GB *dataset*	48	48

➢ Default values for mappers and reducers were used in Hadoop.

➢ Performance evaluation

- According to the experiments conducted shared memory sort take the most time compared to other sorts.
- For Hadoop sort we used default no. of mappers and reducers taken by the system. Hadoop sort performed the best compared to all the other sort.
- Spark sort took more time as compared to Hadoop, as we were only able to use the following specification while running the experiment driver-memory 4g, executor-memory 2g and executor-cores 1.
- Changing the following fields led to spark sort running in an infinite loop probably due to some misconfiguration. But in order to complete the experiments we used the given values, therefore spark sort took more time as compared to Hadoop.
- For Spark sort, we could not validate the output result on valsort, even though the file was sorted. We compared the output of spark sort to output

of linux sort and it was the same. We even tried replacing the newline character with carriage return and newline, but it gave the same results.



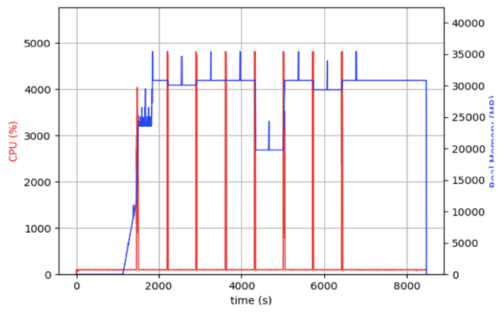
```
cc@team3-new-3:~/gensort-1.5$ cat lsort.txt | head -n 5
    "O!uve  0000000000000000000000001228D4  7777888800002222444 4DDDDDDDEEEE00000000CCCC7777DDDD
   ,K4a-:v  0000000000000000000000001B8132  5555EEEE88889999444 4FFFF1111CCCCEEEE1111EEEE6666FFFF
   .FuD\}u  000000000000000000000000797631  5555DDDDBBBB0000777 72222111122224444DDDDDDDD99996666
   ;5YThct  0000000000000000000000007D3DF5  2222AAAACCCCFFFFAAA A44445555EEEE44442222DDDD99992222
   =2G^9{-  0000000000000000000000000809EE  5555DDDD1111CCCC999 9BBBB0000BBBBCCCCFFFFCCCC44443333
cc@team3-new-3:~/gensort-1.5$ cat  sorted-1GB.txt | head -n 5
    "O!uve  0000000000000000000000001228D4  7777888800002222444 4DDDDDDDEEEE00000000CCCC7777DDDD
   ,K4a-:v  0000000000000000000000001B8132  5555EEEE88889999444 4FFFF1111CCCCEEEE1111EEEE6666FFFF
   .FuD\}u  000000000000000000000000797631  5555DDDDBBBB0000777 72222111122224444DDDDDDDD99996666
   ;5YThct  0000000000000000000000007D3DF5  2222AAAACCCCFFFFAAA A44445555EEEE44442222DDDD99992222
   =2G^9{-  0000000000000000000000000809EE  5555DDDD1111CCCC999 9BBBB0000BBBBCCCCFFFFCCCC44443333
```

- Conclusions
  - Which seems to be best at 1 node scale (1 large.instance)?
    - Hadoop Sort
  - Is there a difference between 1 small.instance and 1 large.instance?
    - Yes, there is a difference between 1 samall.instance and 1 large instance. Overall, the time taken sort on 1 large.instance is less compared to the time taken on 1 small.instance.
  - How about 4 nodes (4 small.instance)?
    - Yes, 4 small nodes seems to be more efficient as compared to 1 large node as the time taken to sort is less while using a 4 node.
  - What speedup do you achieve with strong scaling between 1 to 4 nodes?
    - Almost 65 % speedup
  - What speedup do you achieve with weak scaling between 1 to 4 nodes?
    - Almost 10% speedup
  - How many small.instances do you need with Hadoop to achieve the same level of performance as your shared memory sort?
    - 1 small instance is required, as shown by the experiment that hadoop sort requires less time as compared to shared memory sort .
  - 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?
    - 1 small instance is required, as shown by the experiment that spark sort requires less time as compared to shared memory sort
  - 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?
    - By comparing the bare-metal instance performance from HW5 to to the performance of your sort on a large instance through virtualization, we can see that Shared memory sort takes less time on bare-metal (i.e. for 1GB HW5 takes 87.21 sec and HW6 takes 92.11 sec) whereas linux sort more less time on bare-metal (i.e for 1GB HW5 takes 50.03 sec and HW6 takes 43.72 sec)
  - Can you predict which would be best if you had 100 small.instances?
    - Spark sort would be better with 100 small.instance the memory available will be more and as compared to Hadoop spark does not

need to read and write data to HDFs constantly. It processes data in RAM using Resilient Distributed Dataset, so it will be more efficient.

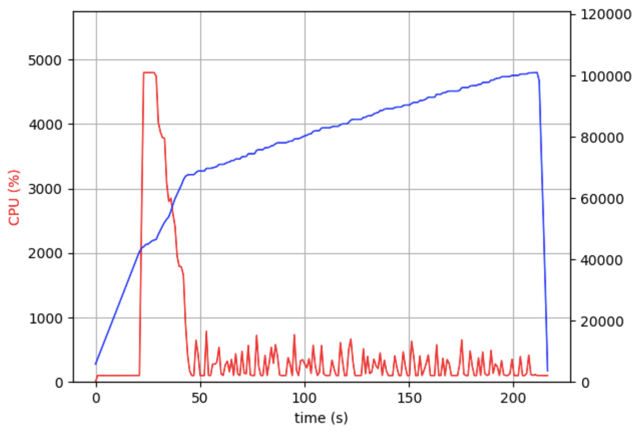
- How about 1000?
    - Even for 1000 small.instances, we belive that spark would be better for the same reasons.
- Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark.
    - Spark won in 2014 for Gray Daytona with given configuration: 207 Amazon EC2 i2.8xlarge nodes x (32 vCores - 2.5Ghz Intel Xeon E5-2670 v2, 244GB memory, 8x800 GB SSD)
    - Hadoop won in 2013 for Gray Dayton with given configuration: 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks)
    - If we compare the results it seems that Sparks work better for less no. of nodes(8) whereas Hadoop works better for large no. of nodes(2100)
    - In that case it seems that for 100 nodes spark would be better whereas for 1000 nodes Hadoop would be better.
- What can you learn from the CloudSort benchmark?
    - It proposes a benchmark using cloud for measuring the efficiency of external sort using public clouds.
    - The benchmarks were: sorting fixed data of 100 bytes records with 10byte key, storing the input and output on a persistent file system and all the operations where performed on commercial public clouds.
    - As the public clouds are not properly provisioned for IO intensive computing and prolonged used of public clouds can be expensive, Cloud sort benchmark would encourage cloud platform to provide support for IO intensive computing and help in moving the enterprises towards using public cloud.
    - External sort was used for the benchmark as it has the most IO-intensive workload. The workload exercises CPU, memory, IO, storage, OS and filesystem.
    - As this benchmark is performed on public cloud it is accessible to everyone and it is affordable.

➢ Graph MySort

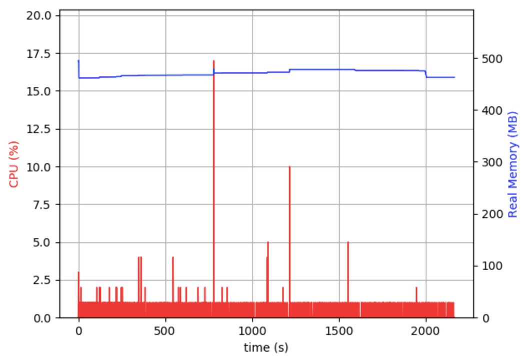


- Graph linux

  Red line – Cpu
  Blue line – memory
  Right side for memory



- Graph Spark 32GB 1 large.instance


CPU (%)
time (s)
Real Memory (MB)