CS553 Cloud Computing - Homework 6
Sort on Hadoop/Spark

Suyog Bachhav - sbachhav@hawk.iit.edu - A20466885
Shekhar Kausal - skausal@hawk.iit.edu - A20512180
Sumit Sakarkar - ssakarkar@hawk.iit.edu - A20516976

This assignment aims to run Linux Sort, Hadoop Sort and Spark Sort on multiple Chameleon nodes. We run these sorts on 1 small (4-cores, 8G RAM, 20G disk), 1 large ((16-cores, 32GB ram, 180GB disk) and 4 small instances. File sizes of 3G, 6G, 12G, 24G are sorted and the results are recorded and analyzed. The goal is to compare the performance of Hadoop and Spark on a single node to that of running it on four nodes. Similarly, comparing Linux sort on instances with different configurations.

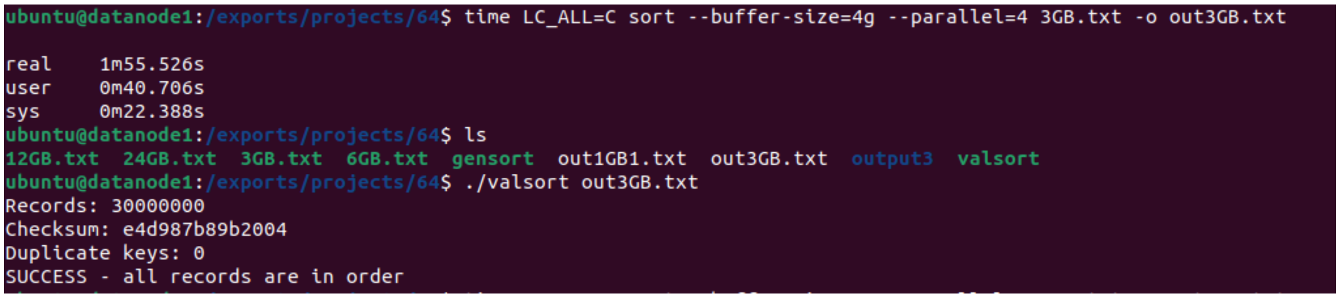
We created and deployed bare-metal instances on Chameleon at CHI@IIT sites and created VMs as per the specified configuration. We set up NFS which was mounted on all the VMs. The ASCII input files are generated and validated using gensort and valsort respectively.

The Linux time command was used to calculate the runtime for each of the sorts and pidstat was used to record information about CPU utilization, memory utilization and I/O speed.

Experiment	**Linux(sec)**	**Hadoop Sort(sec)**	**Spark Sort(sec)**
1 small.instance, 3GB dataset	115.499	322.198	290.37
1 small.instance, 6GB dataset	345.797	812.212	751.62
1 small.instance, 12GB dataset	560.613	Was not able to run	Was not able to run
1 large.instance, 3GB dataset	49.344	205.112	195.25
1 large.instance, 6GB dataset	296.675	353.742	444.95

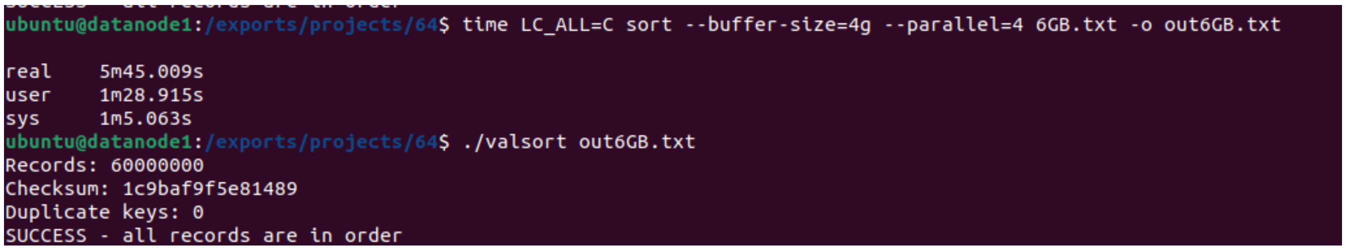
1 large.instance, 12GB dataset	631.08	875.643	800.17
1 large.instance, 24GB dataset	1130.894	2525.942	2100.56
4 small.instances, 3GB dataset	N/A	233.120	190.36
4 small.instances, 6GB dataset	N/A	487.45	396.45
4 small.instances, 12GB dataset	N/A	526.276	341.14
4 small.instances, 24GB dataset	N/A	1229.678	855.32

Small Instance:
Linux Sort
3GB



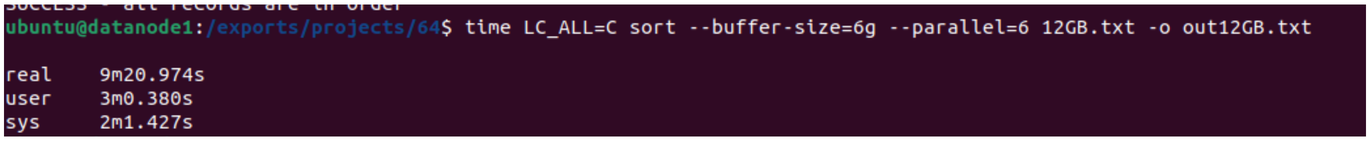
```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort --buffer-size=4g --parallel=4 3GB.txt -o out3GB.txt

real    1m55.526s
user    0m40.706s
sys     0m22.388s
ubuntu@datanode1:/exports/projects/64$ ls
12GB.txt  24GB.txt  3GB.txt  6GB.txt  gensort  out1GB1.txt  out3GB.txt  output3  valsort
ubuntu@datanode1:/exports/projects/64$ ./valsort out3GB.txt
Records: 30000000
Checksum: e4d987b89b2004
Duplicate keys: 0
SUCCESS - all records are in order
```

6GB

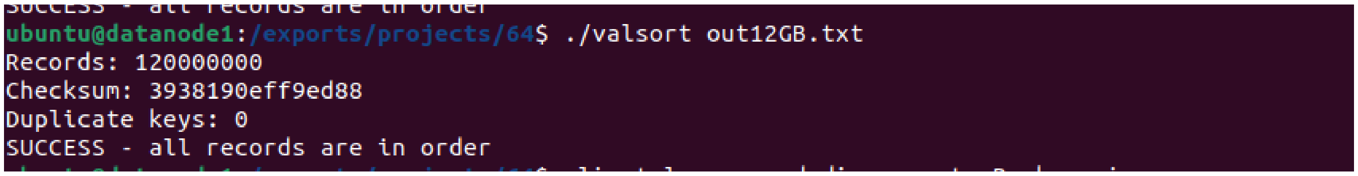


```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort --buffer-size=4g --parallel=4 6GB.txt -o out6GB.txt

real    5m45.009s
user    1m28.915s
sys     1m5.063s
ubuntu@datanode1:/exports/projects/64$ ./valsort out6GB.txt
Records: 60000000
Checksum: 1c9baf9f5e81489
Duplicate keys: 0
SUCCESS - all records are in order
```

12GB:



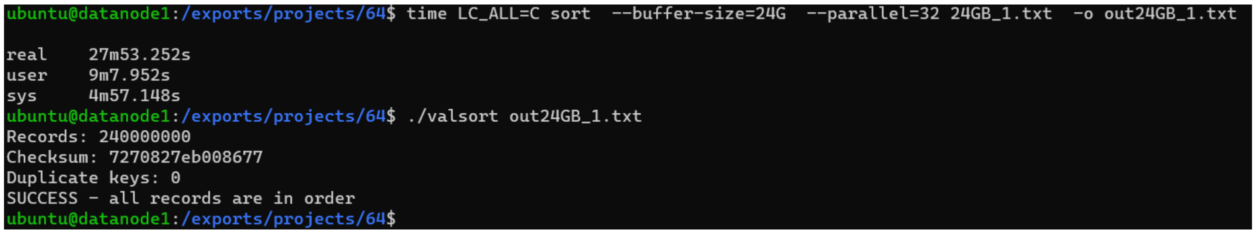
```
SUCCESS - all records are in order
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort --buffer-size=6g --parallel=6 12GB.txt -o out12GB.txt

real    9m20.974s
user    3m0.380s
sys     2m1.427s
```



```
SUCCESS - all records are in order
ubuntu@datanode1:/exports/projects/64$ ./valsort out12GB.txt
Records: 120000000
Checksum: 3938190eff9ed88
Duplicate keys: 0
SUCCESS - all records are in order
```

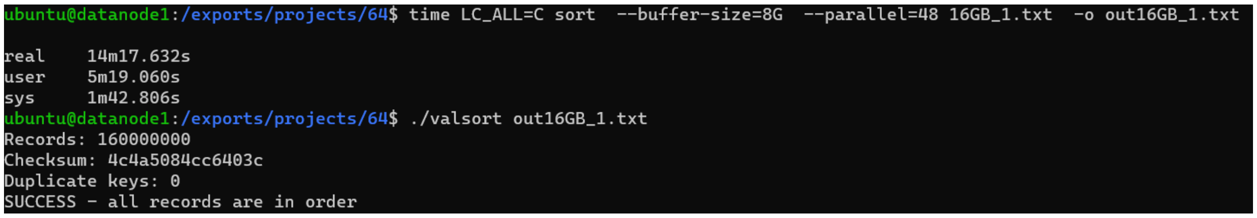
Large instance:

24GB:

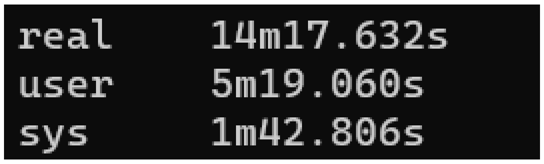


```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort  --buffer-size=24G  --parallel=32 24GB_1.txt  -o out24GB_1.txt

real    27m53.252s
user    9m7.952s
sys     4m57.148s
ubuntu@datanode1:/exports/projects/64$ ./valsort out24GB_1.txt
Records: 240000000
Checksum: 7270827eb008677
Duplicate keys: 0
SUCCESS - all records are in order
ubuntu@datanode1:/exports/projects/64$
```

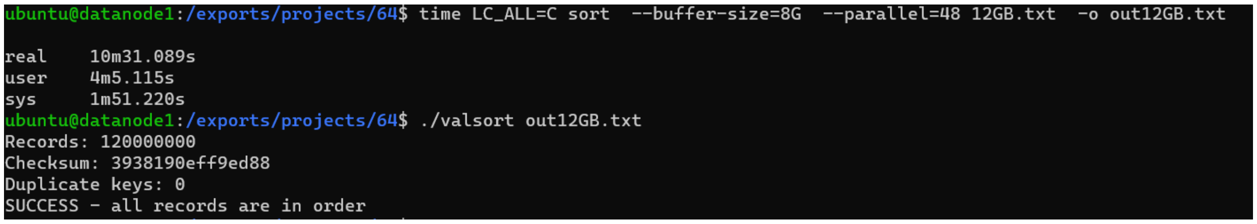
12GB:



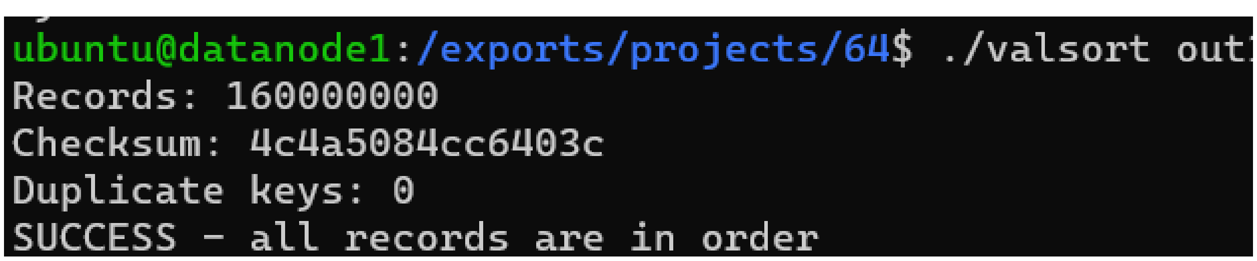
```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort  --buffer-size=8G  --parallel=48 16GB_1.txt  -o out16GB_1.txt

real    14m17.632s
user    5m19.060s
sys     1m42.806s
ubuntu@datanode1:/exports/projects/64$ ./valsort out16GB_1.txt
Records: 160000000
Checksum: 4c4a5084cc6403c
Duplicate keys: 0
SUCCESS - all records are in order
```



```
real    14m17.632s
user    5m19.060s
sys     1m42.806s
```

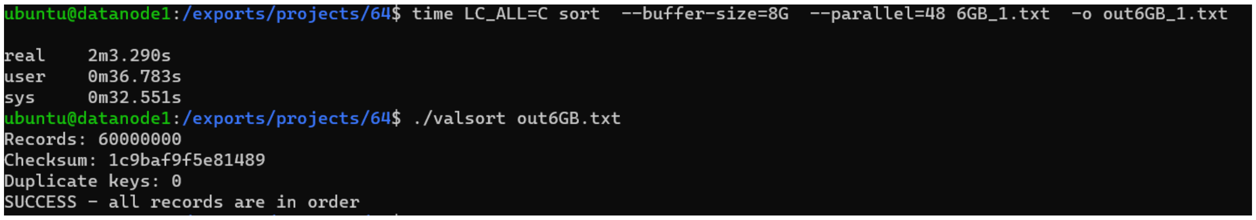


```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort  --buffer-size=8G  --parallel=48 12GB.txt  -o out12GB.txt

real    10m31.089s
user    4m5.115s
sys     1m51.220s
ubuntu@datanode1:/exports/projects/64$ ./valsort out12GB.txt
Records: 120000000
Checksum: 3938190eff9ed88
Duplicate keys: 0
SUCCESS - all records are in order
```



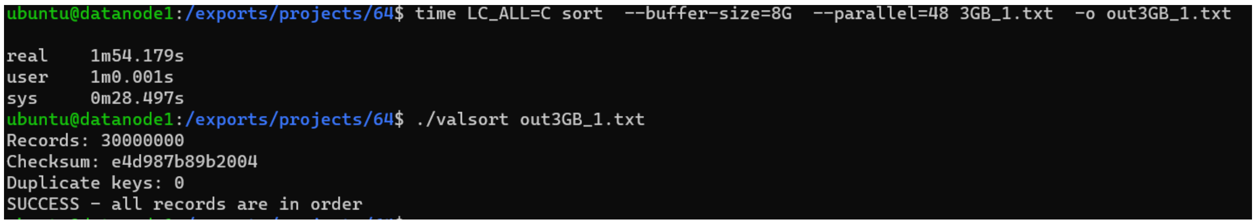
```
ubuntu@datanode1:/exports/projects/64$ ./valsort out
Records: 160000000
Checksum: 4c4a5084cc6403c
Duplicate keys: 0
SUCCESS - all records are in order
```

6GB:

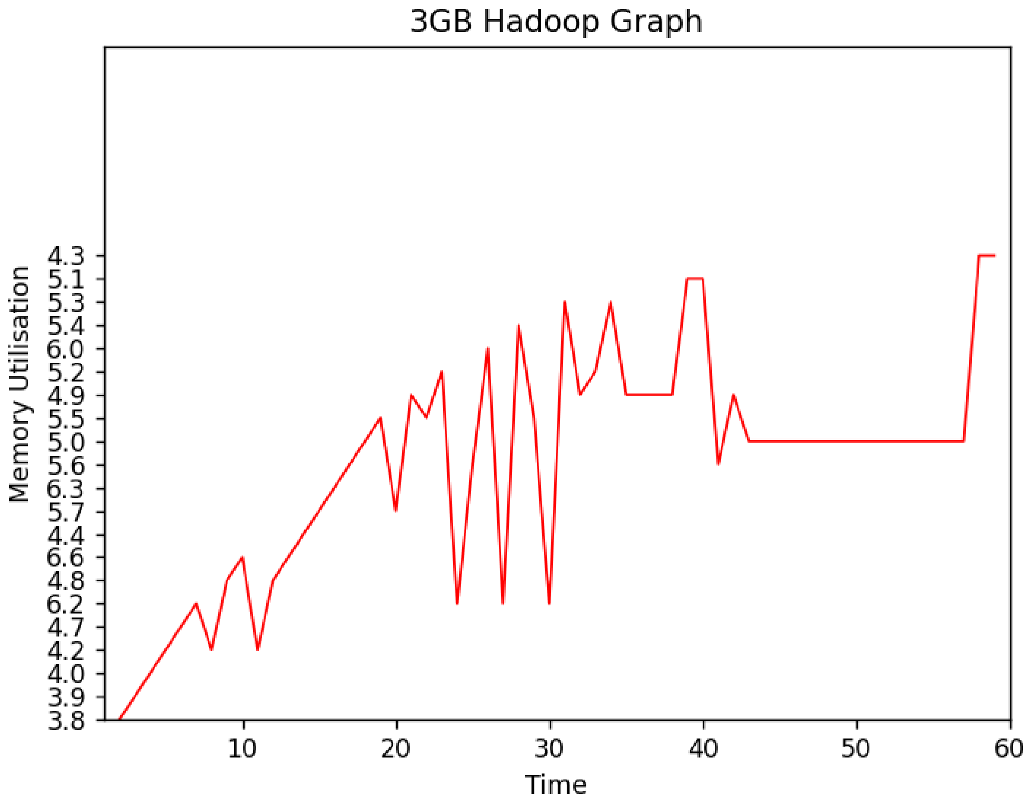


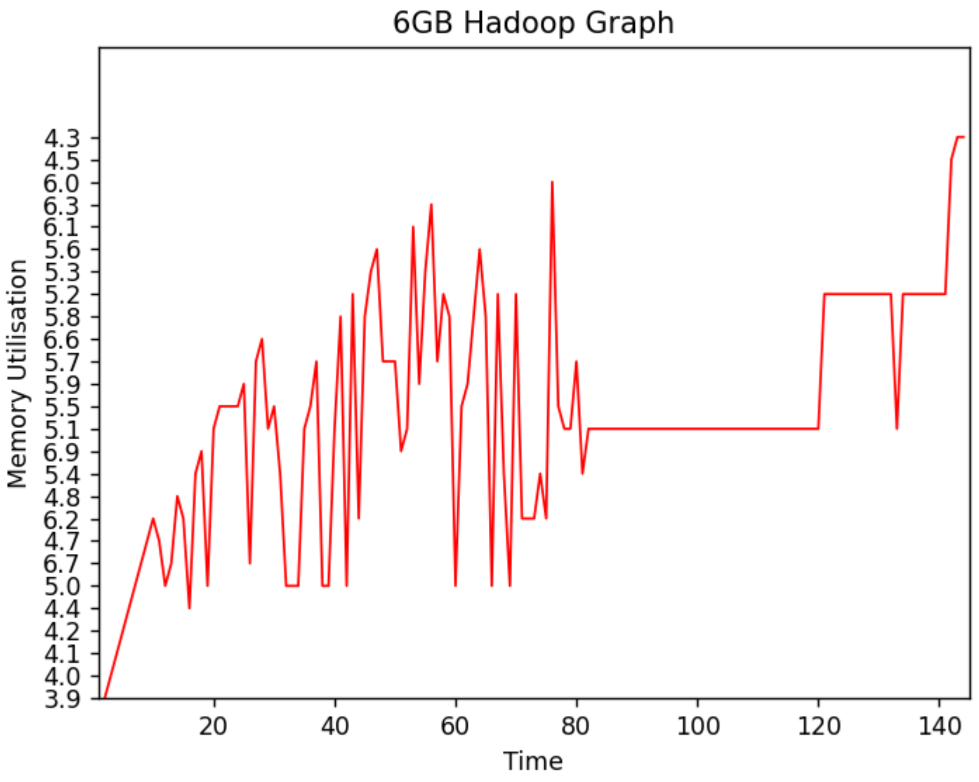
```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort  --buffer-size=8G  --parallel=48 6GB_1.txt  -o out6GB_1.txt

real    2m3.290s
user    0m36.783s
sys     0m32.551s
ubuntu@datanode1:/exports/projects/64$ ./valsort out6GB.txt
Records: 60000000
Checksum: 1c9baf9f5e81489
Duplicate keys: 0
SUCCESS - all records are in order
```

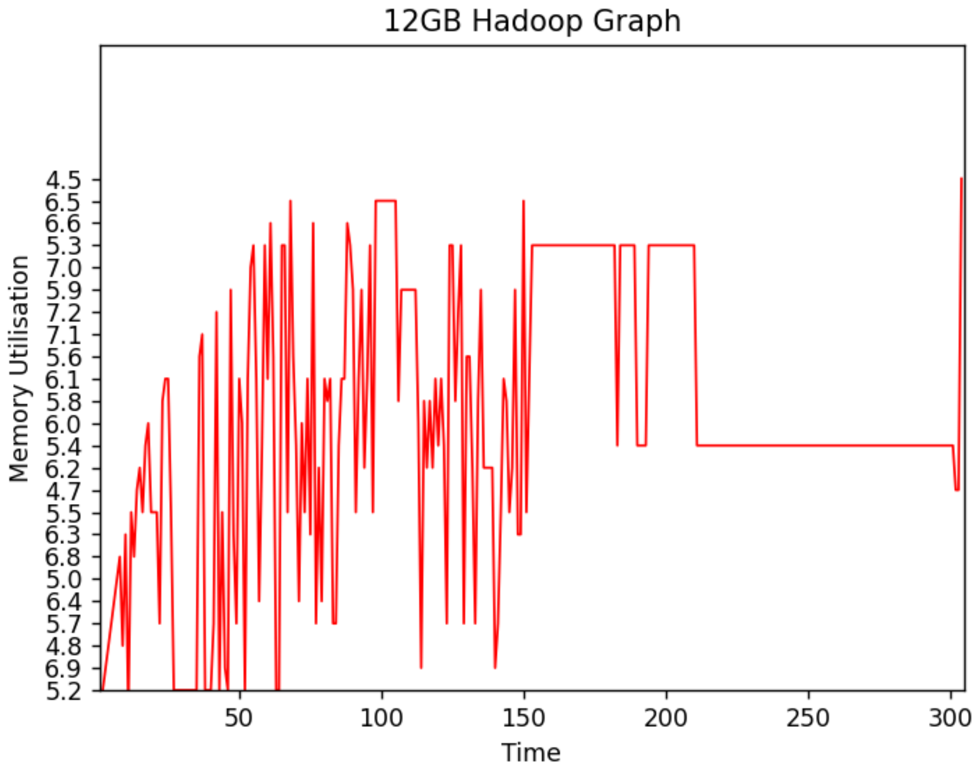
3GB:

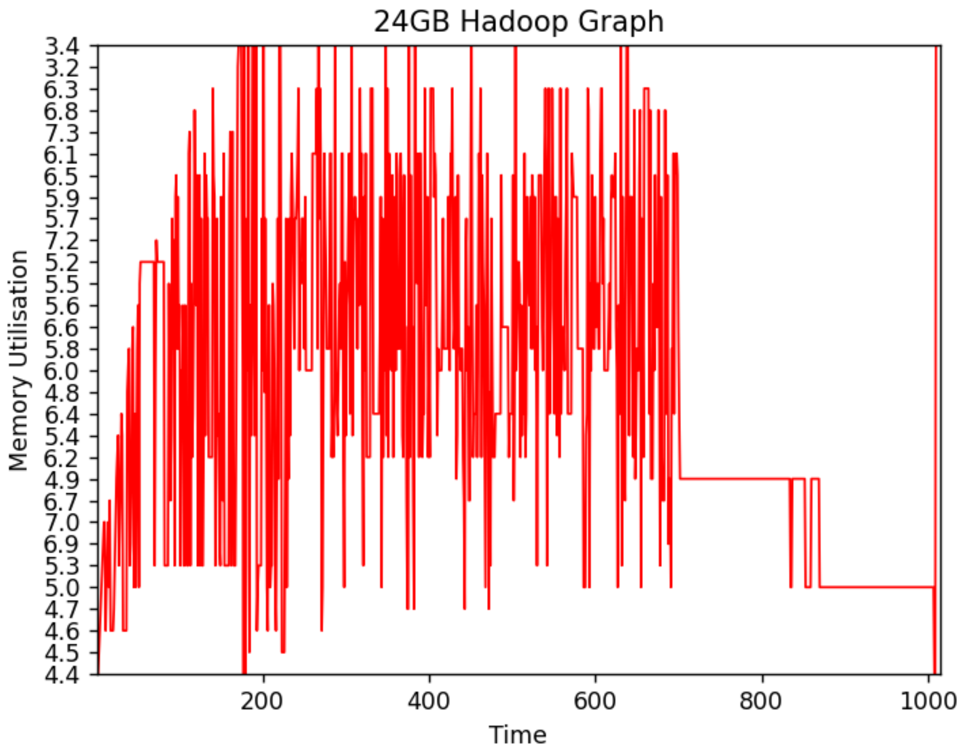


```
ubuntu@datanode1:/exports/projects/64$ time LC_ALL=C sort  --buffer-size=8G  --parallel=48 3GB_1.txt  -o out3GB_1.txt

real    1m54.179s
user    1m0.001s
sys     0m28.497s
ubuntu@datanode1:/exports/projects/64$ ./valsort out3GB_1.txt
Records: 30000000
Checksum: e4d987b89b2004
Duplicate keys: 0
SUCCESS - all records are in order
```


3GB Hadoop Graph
Memory Utilisation
Time
4.3
5.1
5.3
5.4
6.0
5.2
4.9
5.5
5.0
5.6
6.3
5.7
4.4
6.6
4.8
6.2
4.7
4.2
4.0
3.9
3.8
10
20
30
40
50
60


6GB Hadoop Graph
Memory Utilisation
Time
4.3
4.5
6.0
6.3
6.1
5.6
5.3
5.2
5.8
6.6
5.7
5.9
5.5
5.1
6.9
5.4
4.8
6.2
4.7
6.7
5.0
4.4
4.2
4.1
4.0
3.9
20
40
60
80
100
120
140


12GB Hadoop Graph
Memory Utilisation
4.5
6.5
6.6
5.3
7.0
5.9
7.2
7.1
5.6
6.1
5.8
6.0
5.4
6.2
4.7
5.5
6.3
6.8
5.0
6.4
5.7
4.8
6.9
5.2
50
100
150
200
250
300
Time


24GB Hadoop Graph
Memory Utilisation
3.4
3.2
6.3
6.8
7.3
6.1
6.5
5.9
5.7
7.2
5.2
5.5
5.6
6.6
5.8
6.0
4.8
6.4
5.4
6.2
4.9
6.7
7.0
6.9
5.3
5.0
4.7
4.6
4.5
4.4
200
400
600
800
1000
Time

How many threads, mappers, reducers, you used in each experiment?

For small instances, we used 6 threads for Linux Sort, 1 mappers/reducers for Hadoop Sort.
For large instances, we used 24 threads for Linux Sort, 1 mappers/reducers for Hadoop Sort.

How many times did you have to read and write the dataset for each experiment?

To create the dataset and output of the sorted dataset in the system, Linux sort employed gensort. Due to the fact that we are running all of the tests in memory, there will be a total of 2 writes. Before sorting, read the dataset for Reading My Sort and Linsort; reading is also necessary for validation. As a result, Linsort will need a total of two readings of the dataset.
To create the dataset, move the data into the Hadoop file system, output the sorted dataset, and move the sorted output to /exports/projects, Spark and Hadoop used gensort. As a result, Spark and Hadoop sort are run four times in total. We need to read the dataset twice, once in Spark sort and once in Hadoop, to confirm the sorted data.

What conclusions can you draw?

The average CPU utilization for Linux Sort was significantly greater for the large instance. Hadoop sort frequently displayed excessive CPU usage. While Linux sort gradually rose in memory usage, Hadoop and Spark sort displayed high memory utilization. Hadoop still outperforms Spark sort even with 4 nodes.

Which seems to be best at 1 node scale (1 large.instance)?

Linux sort proved to be the best for 1 large.instance whereas **Hadoop** and **Spark** performed poorly due owing to the overhead.

Is there a difference between 1 small.instance and 1 large.instance?

For Linux sort, the small.instance shows faster sort timings almost always. This demonstrates that the performance was unaffected by the number of cores. The performance of Hadoop and Spark is essentially the same for both instances. However, the large.instance performs a little bit better here due to its larger resources.

How about 4 nodes (4 small.instance)?What speedup do you achieve with strong scaling between 1 to 4 nodes?

Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size.

[1] Comparing a 12GB dataset on 1 largeinstance to a 12GB dataset on 4 small instances for Hadoop Sort, we achieve a speedup efficiency of 44%

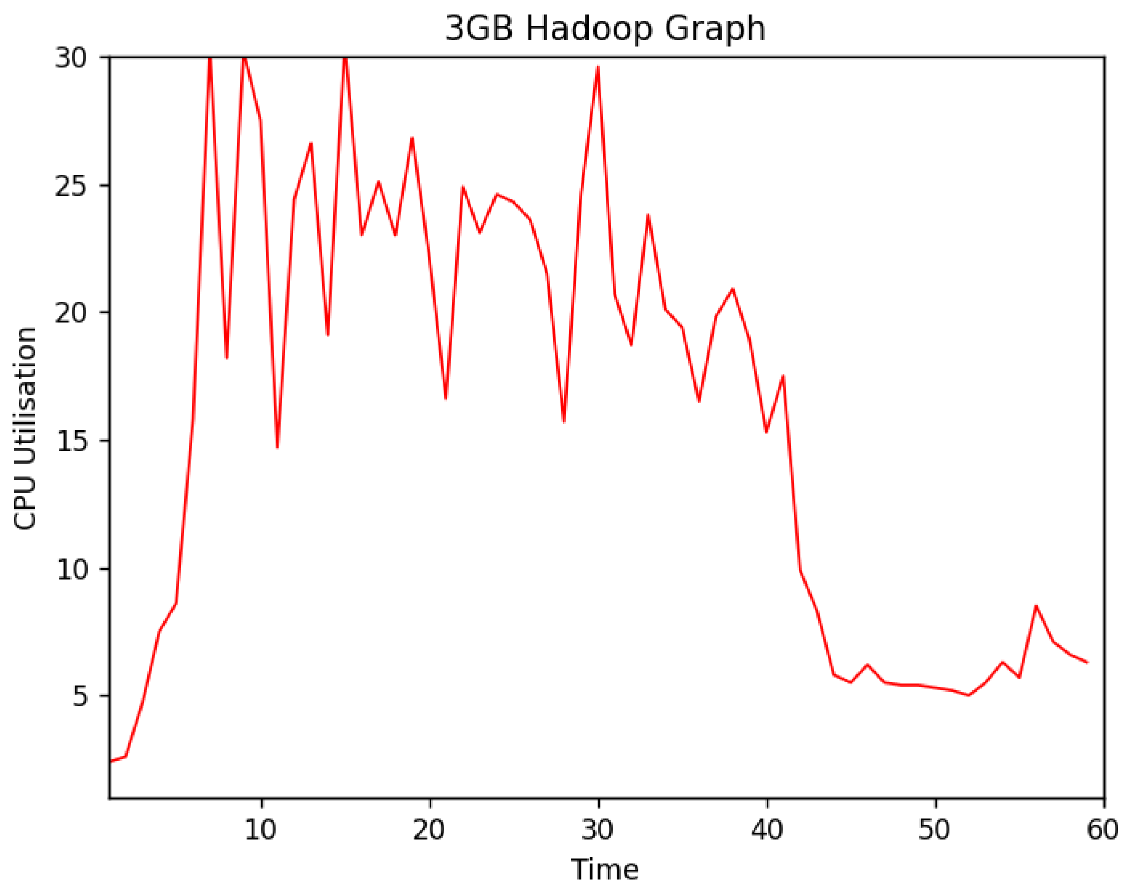
Comparing a 12GB dataset on 1 large instance to a 12GB dataset on 4 small instances for Spark Sort, we achieve a speedup efficiency of 24.98%

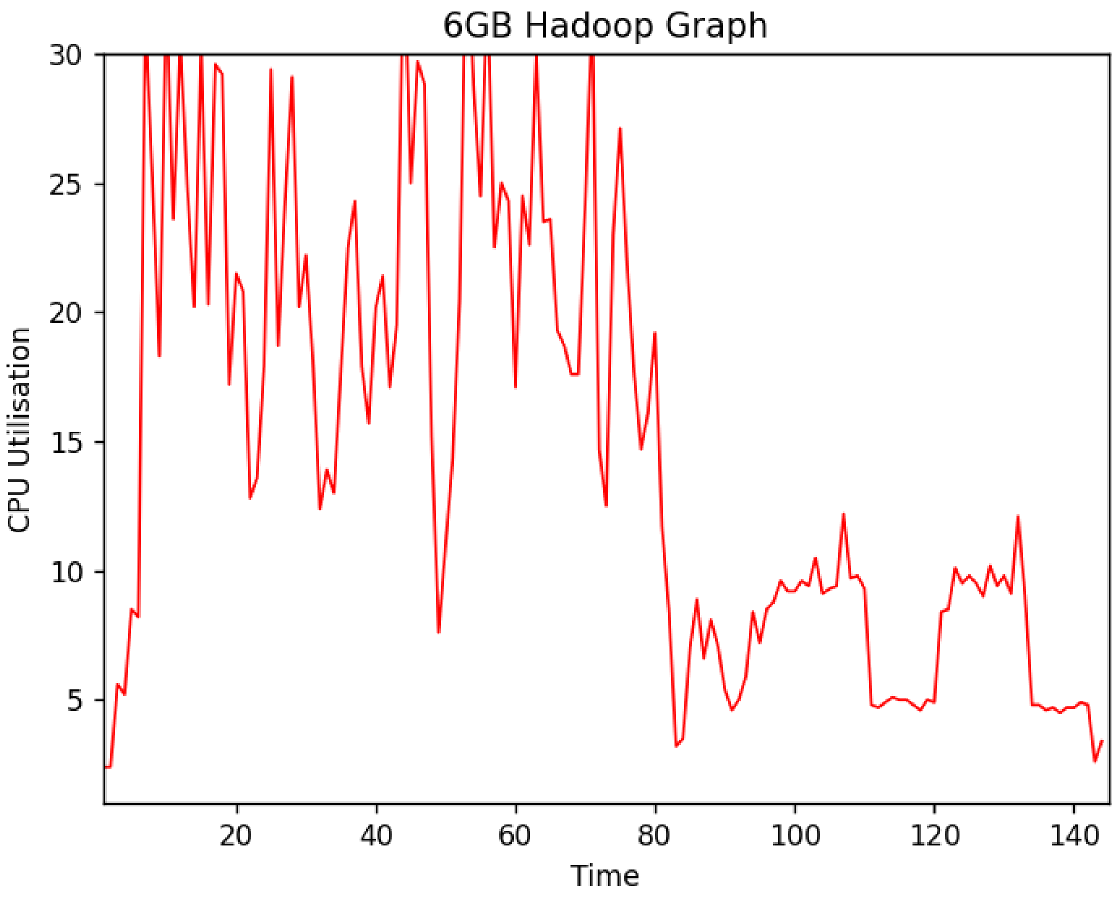
What speedup do you achieve with weak scaling between 1 to 4 nodes?

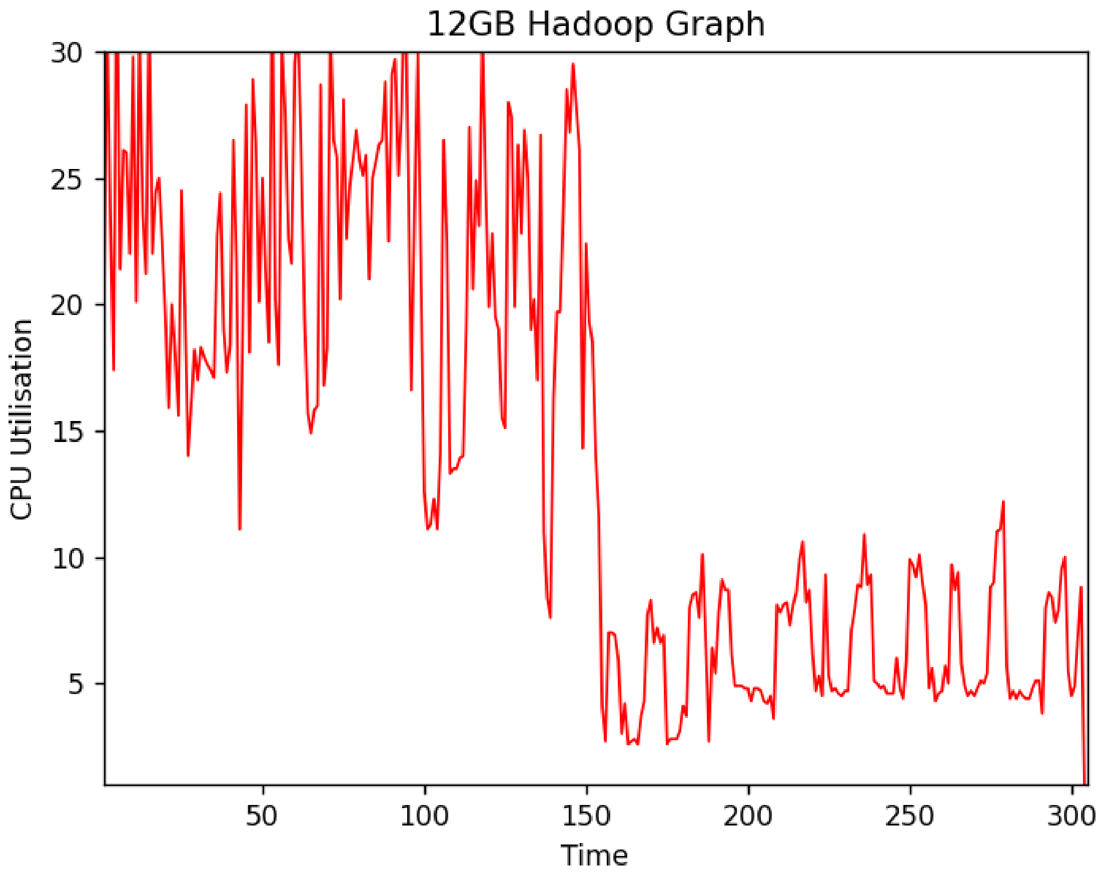
Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

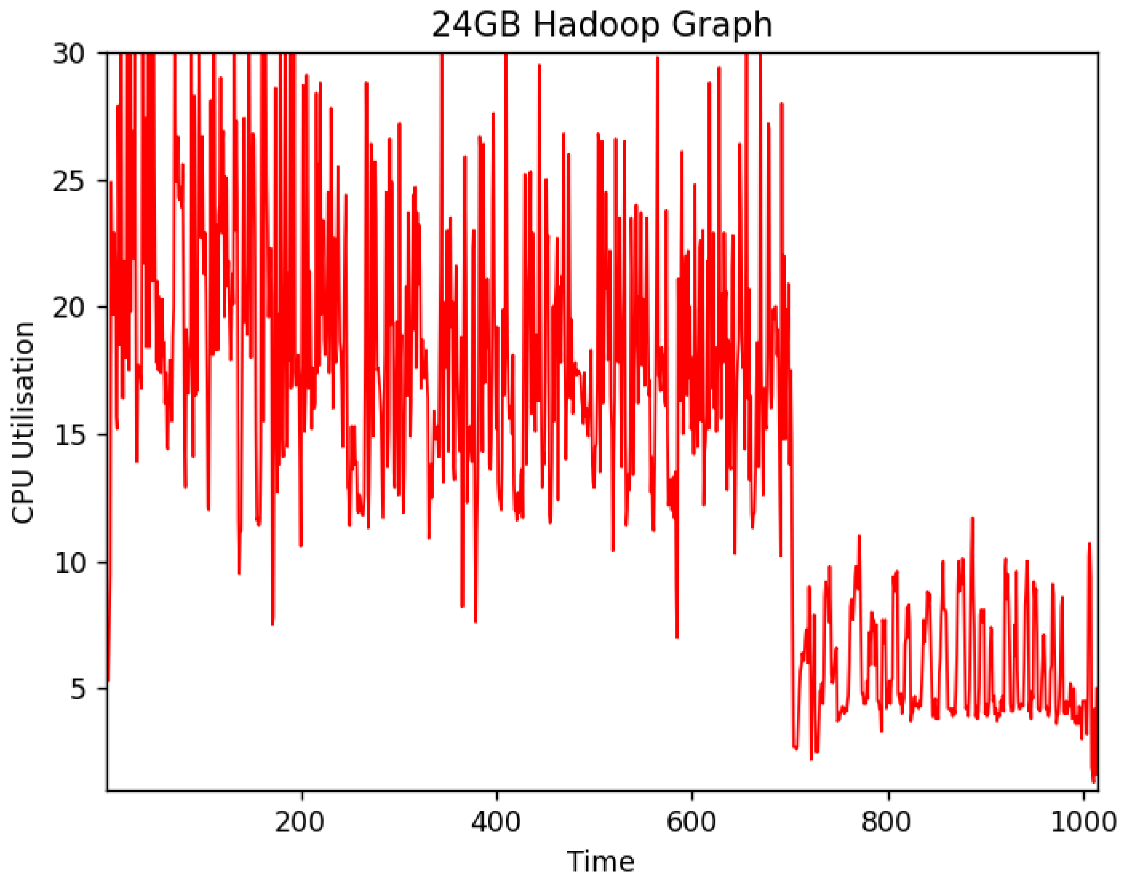
Comparing a 6GB dataset on 1 small instance to a 6GB dataset on 4 small instances for Hadoop Sort, we achieve a speedup efficiency of 51%

Comparing a 6GB dataset on 1 small instance to a 6GB dataset on 4 small instances for Spark Sort, we achieve a speedup efficiency of 54%


3GB Hadoop Graph
CPU Utilisation
Time
30
25
20
15
10
5
10
20
30
40
50
60


6GB Hadoop Graph
CPU Utilisation
Time
30
25
20
15
10
5
20
40
60
80
100
120
140


12GB Hadoop Graph
CPU Utilisation
Time
30
25
20
15
10
5
50
100
150
200
250
300


24GB Hadoop Graph
CPU Utilisation
Time
30
25
20
15
10
5
200
400
600
800
1000