Team members:
Varun Gunda
Allie Bendor
Chandu Chinta

Design Document:

The following strategy is used to sort the input file generated by gensort. If the file size is less than 8 GB (allowed memory size), internal sorting is used. Since if the file size is less than 8 GB, it can fit in the memory and can be sorted using quicksort within memory. This will lead to less costs in terms of time. Hence, in this code, 1 thread is used when the file size is less than 8 GB and internal sorting is used. Since there is only 1 file, it is better to use 1 thread instead of multiple threads since there is only one file to work with. Although we can make the read phase using multiple threads, we did not implement this as we did not see any performance improvement using multiple threads for smaller files. Hence, we went ahead with 1 thread internal sorting implementation if file size is less than 8GB.

However, if the file size is greater than 8 GB, external sorting algorithm is used. This is because the file size is greater than 8 GB and can not be loaded into the memory all at once. Hence we divided the input files into chunks (of size 6 GB *number of threads). Hence we get some **X** as chunk size and we will have file size*/X number of chunks in total. We can sort so many chunks that can fit in memory at once. (Here we limited memory to 6 GB for use by threads since the other 2 GB can be used for stack, code, data segments etc., If we allocate entire 8 GB for threads, then we won't have enough space for other pages.) In every iteration we some sort 6 GB of data and write each chunk to a separate temp file. The process of reading chunks, sorting and writing to temp files is Phase 1. Here for sorting we used quick sort because it performs in place sorting and thus does not require any additional memory. Now all these chunks are merged to form the final sorted output. We used priority queue data structure to build a min heap where each node contains the least string (alphanumeric) and we extract the minimum node, copy it to output "**sorted-data**" file and insert new node in priority queue with the next line from the same file. This way we keep removing nodes and adding nodes until we finish all the chunk files and push the data to output file. This phase 2 is the merge phase of the entire process. This way, we get the sorted output file using external sorting algorithm.

For external sort, if no threads are given as input, 48 threads are chosen by default since from the studies we made, 48 seems to be the best possible threads for external sorting.

Log files:
The command line output of MySort.cpp is used for creating log files. It contains data like read time, write time, sort time etc.,

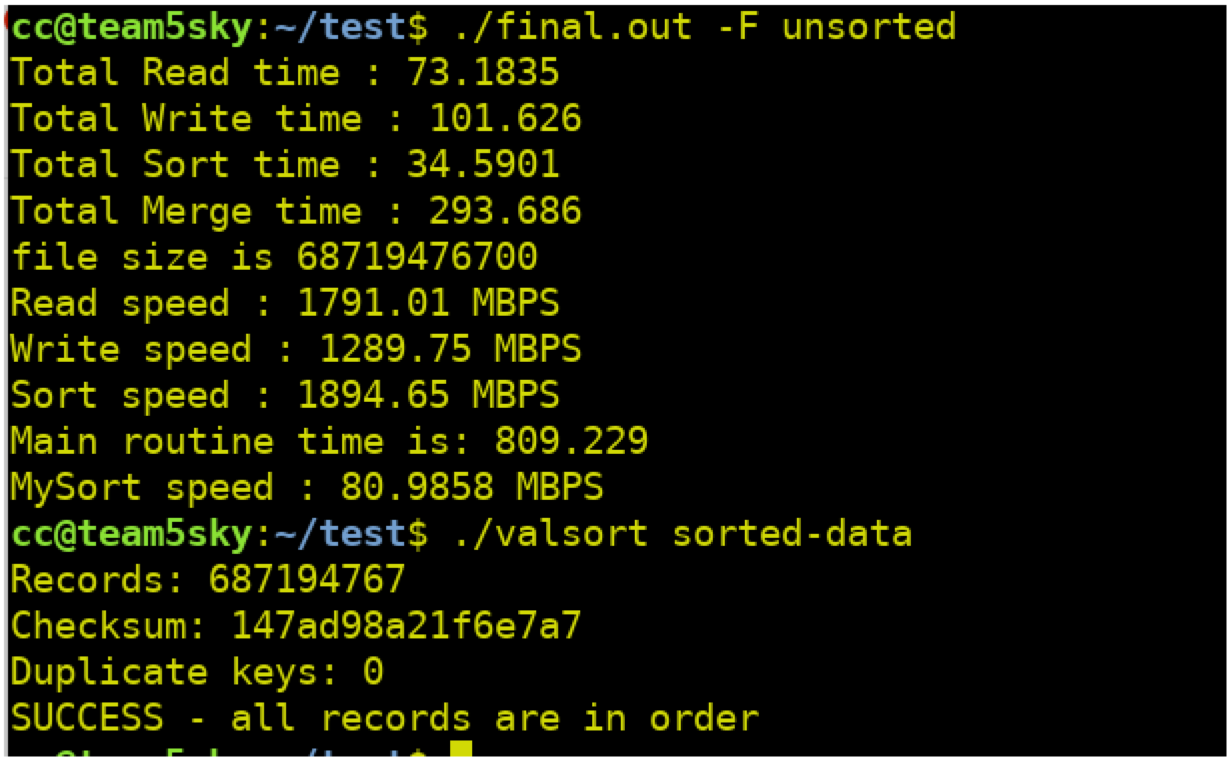
We have included a folder tests, that has all our testing work with multiple file sizes and multiple threads.

Table:

	Shared Memory (1GB)	**Linux Sort (1GB)**	**Shared Memory (4GB)**	**Linux Sort (4GB)**	**Shared Memory (16GB)**	**Linux Sort (16GB)**	**Shared Memory (64GB)**	**Linux Sort (64GB)**
Number of Threads	1	24	1	48	48	48	48	48
Sort Approach	In memory	In memory	In memory	In memory	External	External	External	External
Sort Algorithm	QuickSort	MergeSort	QuickSort	MergeSort	QuickSort + Merge	MergeSort	QuickSort + Merge	MergeSort
Data Read (GB)	1	2	4	8.001	32	32.001	128	128.93
Data Write (GB)	1	1	4	6.47	32	31.59	128	167.58
Sort Time (sec)	33.171	9.95	158.186	63.19	182.365	202.33	812.97	1195.32
Overall I/O Throughput (MB/sec)	298.69	308.58	322.5683	299.72	227.735	253.29	188.5	168.07
Overall CPU Utilization (%)	93.27	222	98.183	276.9	352.146	244.256	312.51	210.67
Average Memory Utilization (GB)	1.165	1.68	4.61	5.27	1.49	7.12	1.32	7.542

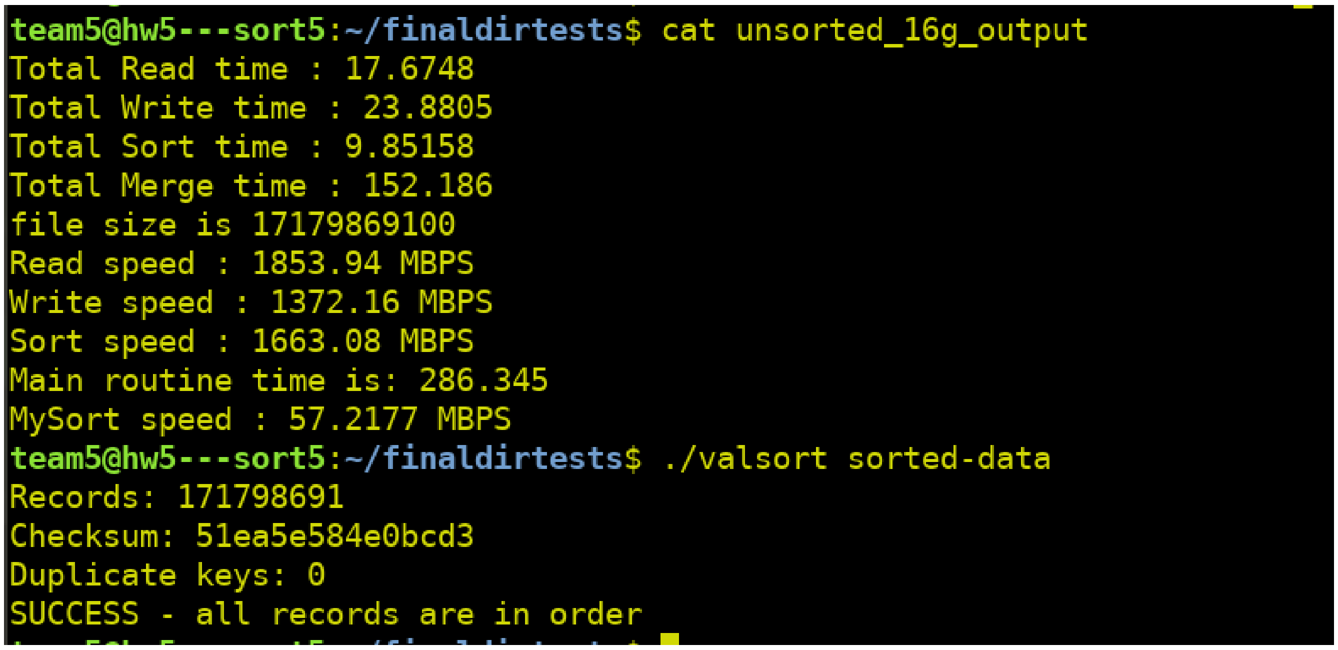
From the table, we can see that the MySort.cpp performs better than linux sort for file sizes 16 GB and 64GB. However, linux sort has an upper edge for the file sizes 1 GB and 4 GB. This is because there is only 1 thread working for MySort.cpp for file sizes less than memory size i.e., 8 GB. There can be some performance improvement that can be made to MySort.cpp i.e., by using multi threads for reading instead of 1 thread. Also, overall CPU utilization is over 100% when we use external sort since multiple threads are used. Also, average memory utilization is very less within 8 GB. This is very less compared to memory used by linux sort.. The overall I/O throughput is around 250 MBPS most of the time. The sort algorithm used is quick sort with merge in the phase 2 to merge all the chunk files. We tried testing mysort with different number of threads and figured out that 48 threads is the best. We included all our observations in the tests directory. This 48 threads also makes sense since there are 48 logical cores on the skylake machine and hence one thread can run on one logical core and result in best performance compared to lesser number of threads.

Testing mysort on 64 GB file:



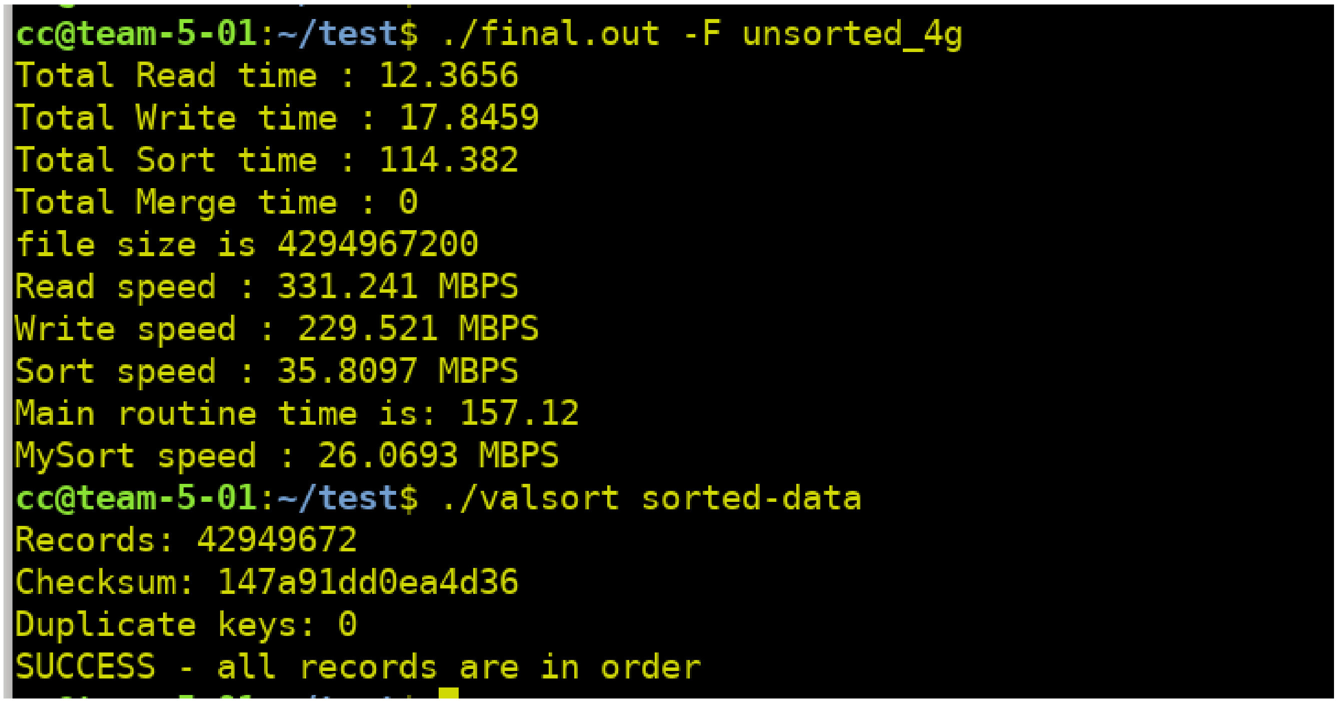
```
cc@team5sky:~/test$ ./final.out -F unsorted
Total Read time : 73.1835
Total Write time : 101.626
Total Sort time : 34.5901
Total Merge time : 293.686
file size is 68719476700
Read speed : 1791.01 MBPS
Write speed : 1289.75 MBPS
Sort speed : 1894.65 MBPS
Main routine time is: 809.229
MySort speed : 80.9858 MBPS
cc@team5sky:~/test$ ./valsort sorted-data
Records: 687194767
Checksum: 147ad98a21f6e7a7
Duplicate keys: 0
SUCCESS - all records are in order
```

Testing on 16gb file:



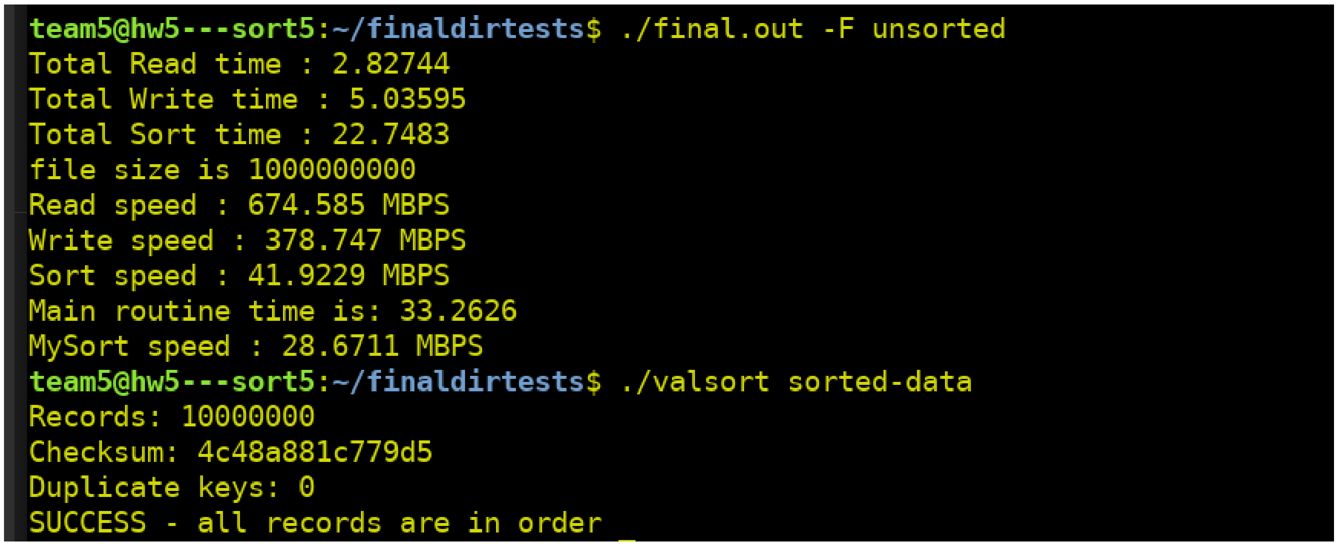
```
team5@hw5---sort5:~/finaldirtests$ cat unsorted_16g_output
Total Read time : 17.6748
Total Write time : 23.8805
Total Sort time : 9.85158
Total Merge time : 152.186
file size is 17179869100
Read speed : 1853.94 MBPS
Write speed : 1372.16 MBPS
Sort speed : 1663.08 MBPS
Main routine time is: 286.345
MySort speed : 57.2177 MBPS
team5@hw5---sort5:~/finaldirtests$ ./valsort sorted-data
Records: 171798691
Checksum: 51ea5e584e0bcd3
Duplicate keys: 0
SUCCESS - all records are in order
```

Testing on 4 gb file:



```
cc@team-5-01:~/test$ ./final.out -F unsorted_4g
Total Read time : 12.3656
Total Write time : 17.8459
Total Sort time : 114.382
Total Merge time : 0
file size is 4294967200
Read speed : 331.241 MBPS
Write speed : 229.521 MBPS
Sort speed : 35.8097 MBPS
Main routine time is: 157.12
MySort speed : 26.0693 MBPS
cc@team-5-01:~/test$ ./valsort sorted-data
Records: 42949672
Checksum: 147a91dd0ea4d36
Duplicate keys: 0
SUCCESS - all records are in order
```

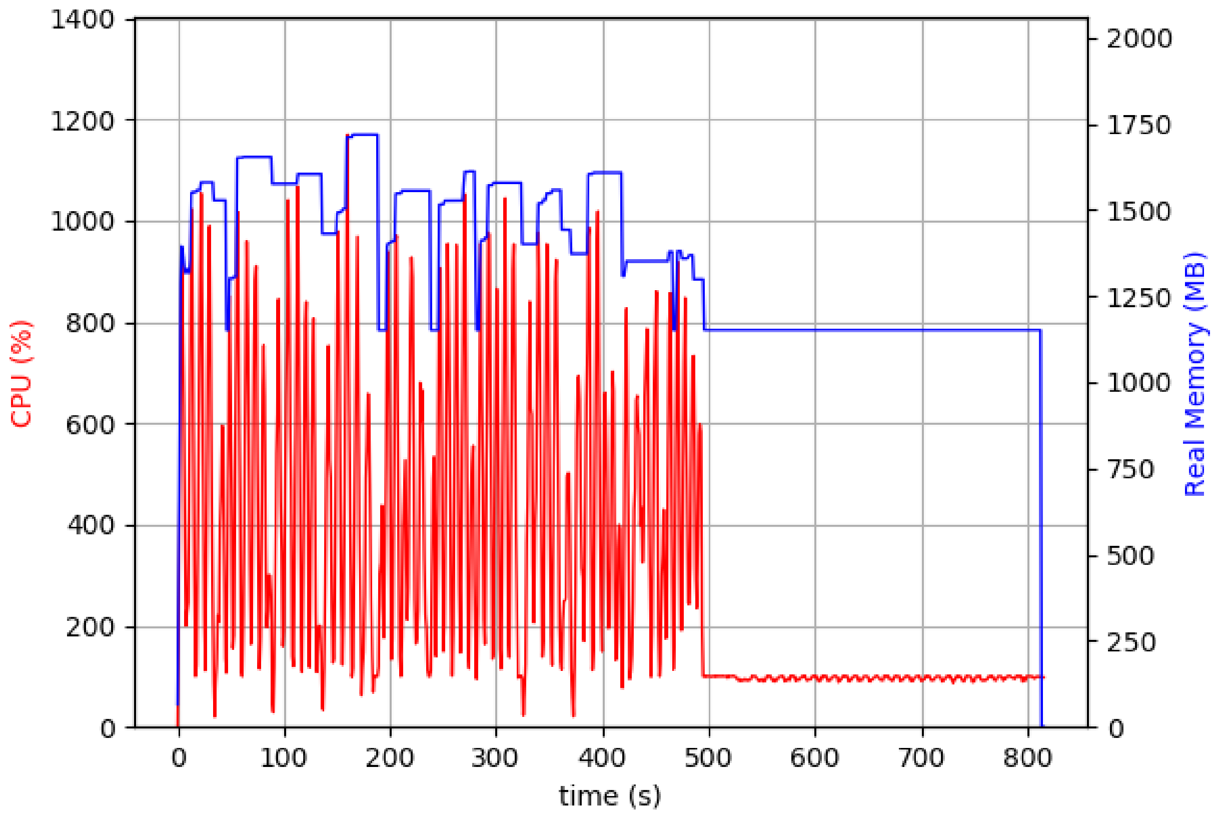
Testing on 1 gb file:



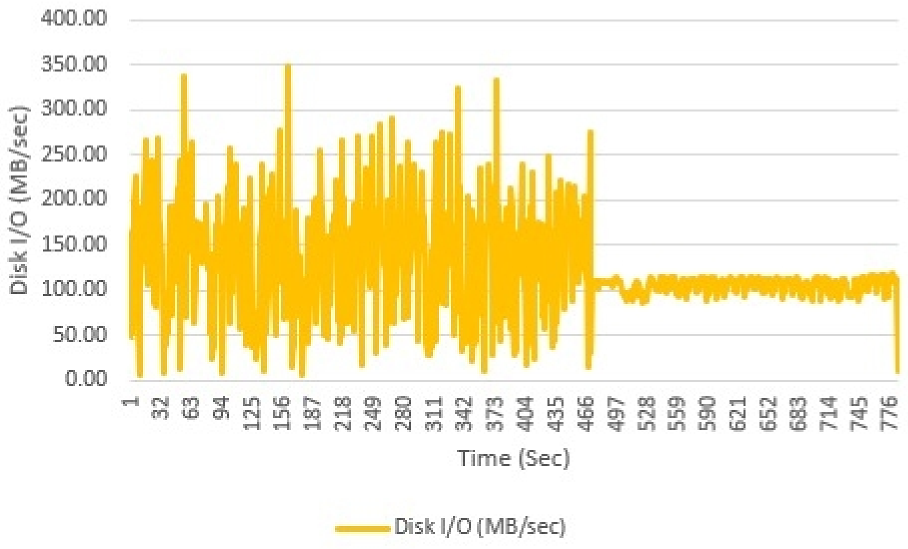
```
team5@hw5---sort5:~/finaldirtests$ ./final.out -F unsorted
Total Read time : 2.82744
Total Write time : 5.03595
Total Sort time : 22.7483
file size is 1000000000
Read speed : 674.585 MBPS
Write speed : 378.747 MBPS
Sort speed : 41.9229 MBPS
Main routine time is: 33.2626
MySort speed : 28.6711 MBPS
team5@hw5---sort5:~/finaldirtests$ ./valsort sorted-data
Records: 10000000
Checksum: 4c48a881c779d5
Duplicate keys: 0
SUCCESS - all records are in order
```

As we see in the above images, I tested MySort.cpp on multiple datasets that include 1 GB, 4 GB, 16 GB and 64 GB. Data for each of them is generated .using gensort and after sorting using **"final.out" (**which is the executable created from MySort.cpp), validated using valsort. In all the cases, the sorting algorithm is working perfectly fine as sen by the valsort output.

Plots:



1400
1200
1000
800
600
400
200
0
CPU (%)
2000
1750
1500
1250
1000
750
500
250
0
Real Memory (MB)
0
100
200
300
400
500
600
700
800
time (s)



400.00
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
Disk I/O (MB/sec)
1
32
63
94
125
156
187
218
249
280
311
342
373
404
435
466
497
528
559
590
621
652
683
714
745
776
Time (Sec)
Disk I/O (MB/sec)