Problem:

sorting using in-memory and external sorting in a single node. The programs help to solve vast amounts of data by using external sorting and smaller datasets on memory. I used gen sort to generate the data and valsort to verify it.

Note:: logs are only generated when the program executes and linsort logs can be generated using the gen sort data files

Methodology :

1) First implemented basic smart logic for two sorts types
2) When file size is greater than 8GB it executes external sorting else in-memory sort
3) I have implemented read-write IO for reading writing files first
4) Then implement parallel merge sort
5) This sorting is based on key values generated by the gen sort
6) Then, I implemented external sorting first by merging and splitting files
7) It will do in-memory sorting using parallel merge sort and the data written into a temp file
8) Then using min heap will merge all files in a sorted order
9) It will take time to create temp files because of heavy IO operations

Here are the implementations of the functions with short details

void write_output_file(char **keys, int actual_keys, const char *outputFile) - Writes the sorted keys to an output file.

void mergeSort(char **arr, int l, int r) - Implements the merge sort algorithm on an array of strings.

void merge(char **arr, int l, int m, int r) - Merges two subarrays within the array for the merge sort.

void read_file(const char *filename, char ***keys, int *num_keys) - Reads a file containing keys and stores them in an array of strings.

unsigned long long getFileSize(const char *filename) - Retrieves the size of a file in bytes.

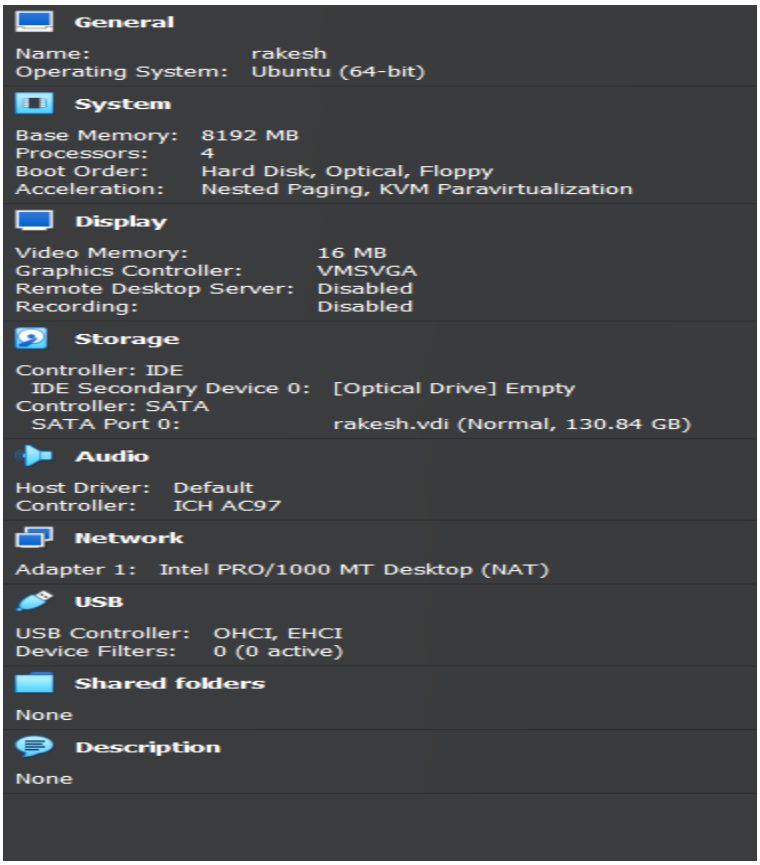
void* threadedMergeSort(void* arg) - A thread function for performing merge sort on a portion of the array.

void parallelMergeSort(char **arr, int l, int r, int max_threads) - Initiates parallel merge sort using multiple threads.

void validate_sort(const char *output_filename) - Executes an external program to validate if the output file is correctly sorted.

void merge_temp_files(int num_temp_files, const char *output_file) - Merges temporary files generated during external sorting into a single output file.

void remove_empty_lines(const char *filename) - Removes empty lines from a text file.



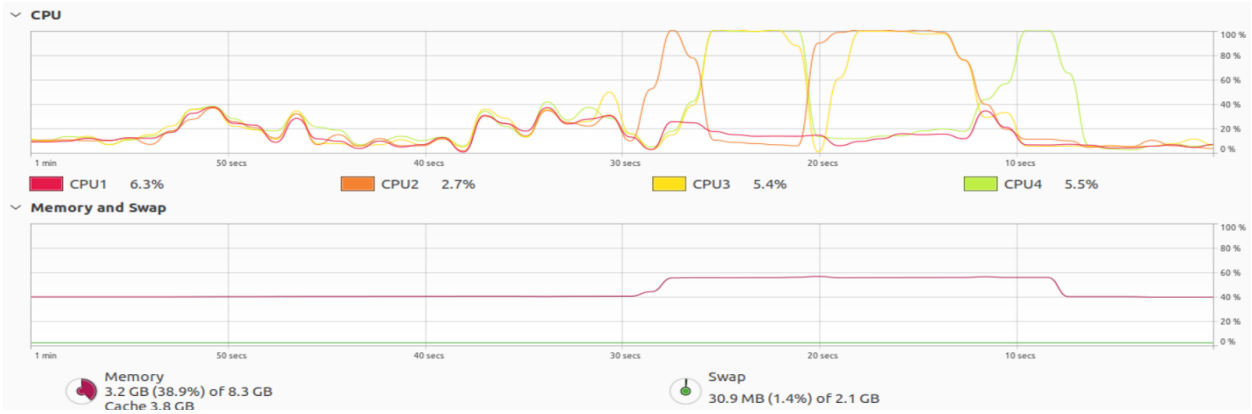
Above are my system configurations

Performance analysis

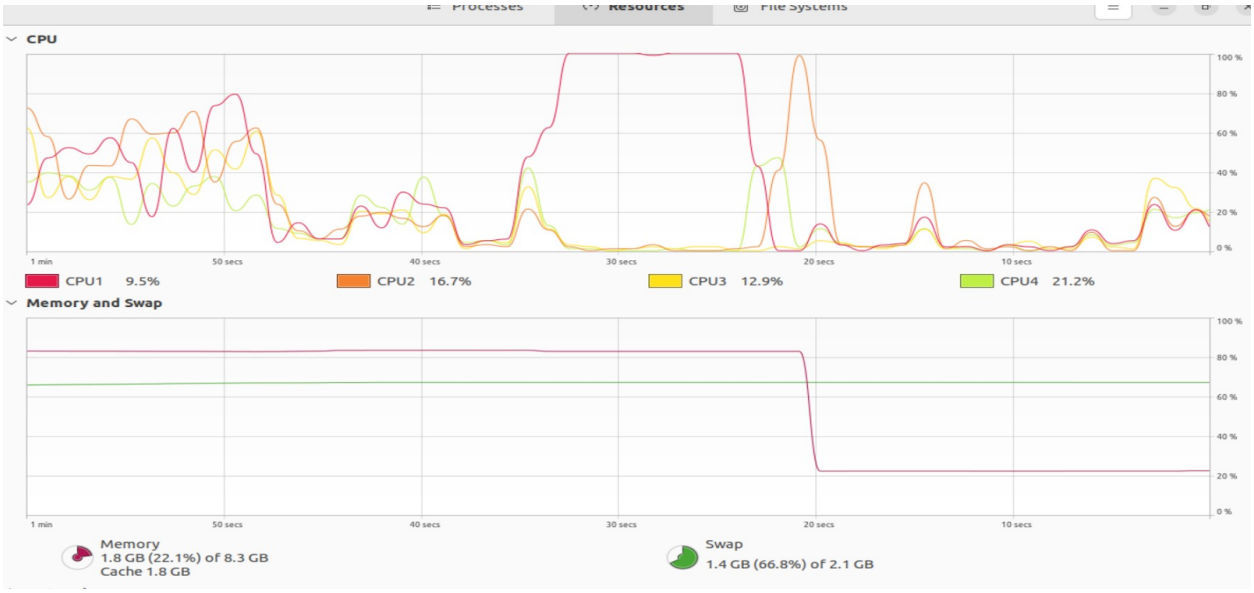
Experiment	Shared Memory (1GB)	Linux Sort (1GB)	Shared Memory (4GB)	Linux Sort (4GB)	Shared Memory (16GB)	Linux Sort (16GB)	Shared Memory (64GB)	Linux Sort (64GB
Number of Threads	4	4	6	6	8 (max)	8	8(max)	8(max)
Sort Approach (e.g. inmemory / external)	in-memory	in-memory	in-memory	External sorting	External sorting	External sorting	External sorting	External sorting

Sort Algorithm (e.g. quicksort / mergesort / etc)	merge sort	linsort	Merge sort	linsort	mergesort	linsort	Merge sort	linsort
Data Read (GB)	1GB	1GB	4GB	4GB	32	32 (approximately)	128GB	128
Data Write (GB)	1GB	1GB	4GB	4GB	32	32(approximately)	128GB	128
Sort Time (seconds)	: 14.722	18.291	87.229621	71.931	941.893884	720	3360.182639	3211
Overall I/O Throughput (MB/sec)	67921.138396	53821.358	45855.982797	56818.71276	16987.051590	22762.82672	11987.051590	13532.2343
Overall CPU Utilization (%)	80	68	32	52	86	91	87	Avg of 96

Graph for 4GB data



Graph for 64GB data :

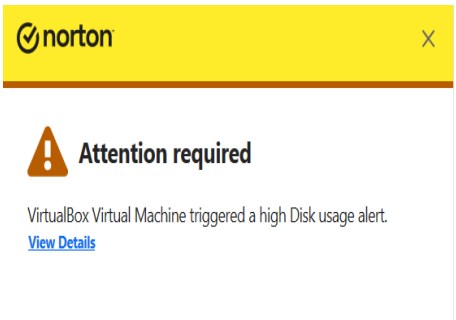


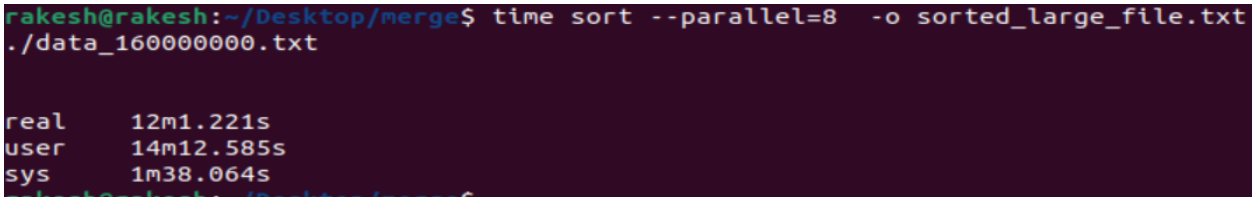
Command use for linsort and generate logs on my gendata files

time sort --parallel=4 -o linsort1GB.log ./data_10000000.txt
time sort --parallel=6 -o linsort4GB.log ./data_40000000.txt
time sort --parallel=8 -o linsort16GB.log ./data_160000000.txt
time sort --parallel=8 -o linsort64GB.log ./data_640000000.txt

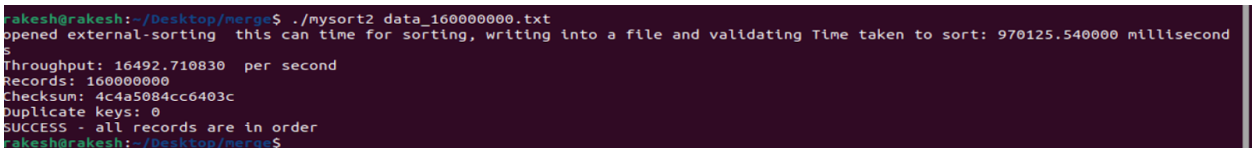
Here some interesting observations on external sorting

- Firstly for external sorting disk usage is high



- 

- The above image is the proof of linsort

- 

- **T**he above image is the proof of external sort

- In terms of sort time, Linux Sort is often quicker than Shared Memory for both 16GB and 64GB datasets. This shows that Linux Sort may be better suited for large-scale external sorting.
- In terms of I/O throughput, Linux Sort routinely beats the Shared Memory method. Higher throughput in Linux Sort may suggest more efficient disk I/O operations.
- Linux Sort has higher CPU consumption, particularly for the 64GB dataset (98%). This shows that Linux Sort may be more CPU-bound than Shared Memory, thus optimizing CPU cycles better.

Here some interesting observations on in-memory sorting on my results.

- When compared to the Shared Memory technique, Linux Sort is slower for 1GB data but quicker for 4GB data. This implies that the Shared Memory technique may be more efficient for smaller data sets, but Linux Sort may be better suited for bigger data sets.

- The Shared Memory technique offers greater I/O throughput with 4 threads for both 1GB and 4GB, whereas Linux Sort has higher throughput with 6 threads. This shows that when employing more threads, Linux Sort may be more I/O-efficient, especially for bigger datasets.

- When compared to Linux Sort, the Shared Memory method has greater CPU consumption for 1GB data but much-reduced CPU utilization for 4GB data. Higher CPU usage may signify more efficient use of available resources, but it may also imply less capacity for multitasking.

Challenges faced::

- **Writing code is a level but debugging takes a hell lot of time to solve some issues**
- **Learnt heap sort and implementation took a considerable amount of time**
- **Have faced issues regarding key spaces and empty lines**
- **Many times my VM crashed but redone the work and saved code in doc files**
- **Lastly, i need to wait a lot of time for 64GB of sorting**