PA 3

Ganesh Prasad Chandra Shekar – A20557831

Ramya Venkatesh – A20546964

Experiment	Shared Memory(1GB)	Linux Sort (1GB)	Shared Memory(4GB)	Linux Sort (4GB)	Shared Memory(16GB)	Linux Sort (16 GB)	Shared Memory(64GB)	Linux Sort (64 GB)
Number of Threads	1	4	8	8	8	8	8	8
Sort Approach	In-memory	In-memory	In-memory	In-memory	External	External	External	External
Sort Algorithm	Quick sort	Merge Sort	Quick sort	Merge sort	Quick sort	Quick sort	Quick sort	Quick sort
Data Read (GB)	1 GB	0.25 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread
Data Write (GB)	1 GB	0.25 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread	0.5 GB per thread
Sort Time(sec)	36.408	52.54	86.34	364.70	547.88	1214.69	2307.34	1924.54
Overall I/O Throughput (Mb/sec)	27.466	19.033	46.32	10.967	29.203	13.17	27.73	33.25
Overall CPU Utilization (%)	100	99.1	77.8	157	80.1	143	71.34	139.04
Average Memory Utilization (GB)	1.303	1.46	4.16	0.92	2.47	1.35	3.31	2.78

For smaller data size that is 1GB, 4GB (i.e., while utilizing internal sort). Linux sorting performs better as the sorting algorithm used by Linux sorting is more optimized while our Terasort C program performs poorly mainly due to the lack of code optimization.

For larger data size that is 16GB, 64GB (i.e., while utilizing external sort). The performance of our C program implementation is better when compared to that of Linux sort since the performance of the CPU is good for sorting data.
Memory limit for our VM is 4GB, and in our program we have implemented multi-threaded pool which supports up to 8 threads. For our benchmarking we have utilized thread size of 8.

For dataset size greater than 4GB, we utilize external sort. For the Terasort benchmarking we divide the big file into smaller chunks i.e., temporary files and later merge them together. Here quick sort makes more sense to use since merge sort takes up more space.

Scope for Optimization:

1. We can increase the node/laptop main memory size to a much larger size to accommodate in-memory sorting approach for much larger dataset size.
2. The quick sort algorithm can be coded manually instead of using the qsort() library.

Steps to generate and test the files:

1. Generate datasets using Gensort

Example: ./gensort -a 100000000 inputfile_1GB

This command generates ascii records of size 1GB and generates a output file with the name "inputfile_1GB"

2. To perform Linux sort

```
time sort -k 1 inputfile_1GB -o sorted1GBlinux.txt --parallel=8 > linsort1GB.log 2>&1
```

This command runs sorting on 1GB file named 'inputfile_1GB' and outputs sorted1GBlinux.txt and a log file with name 'linsort1GB.log'

3. To perform C program Shared-Memory Terasort - mysort.c

----

*first compile the code using the command below->

```
        gcc -mysort.c -o mysort
```

*Post compiling, run the executable ->

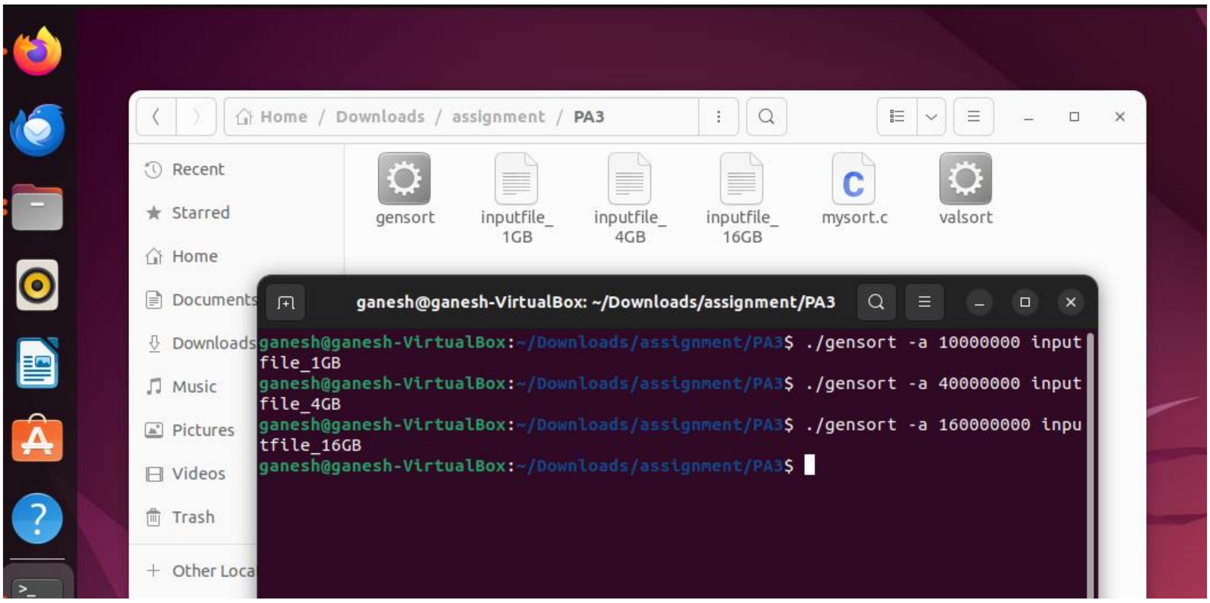
./mysort <input file> <output file> <number of threads>

Example: ./mysort inputfile_1GB sorted1GBmysort.txt 8 >> mysort1GB.log 2>&1

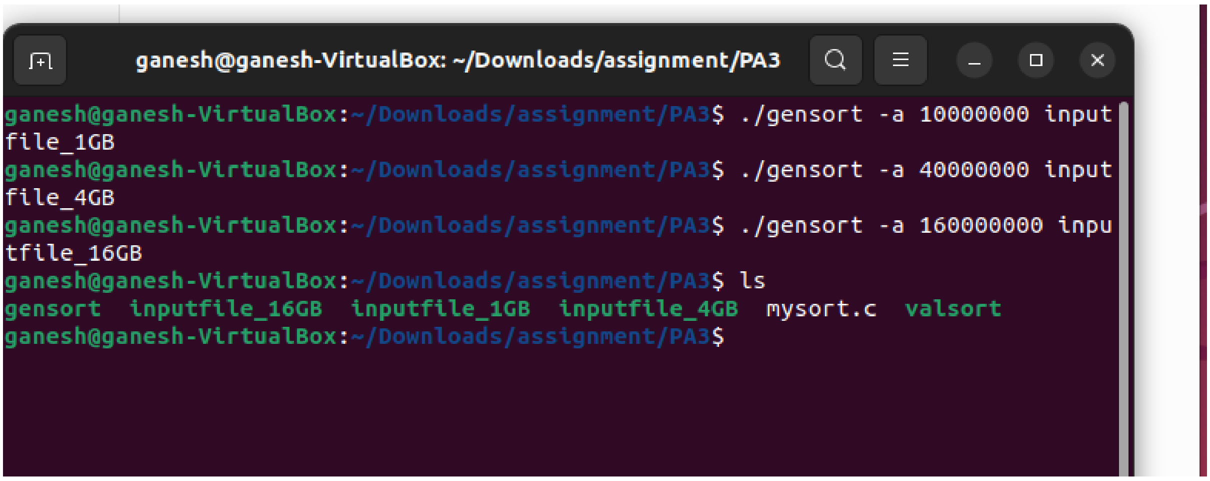
4. To validate using Valsort

Example: ./valsort sortedfilename.txt

Some screenshots of generating file using gensort-



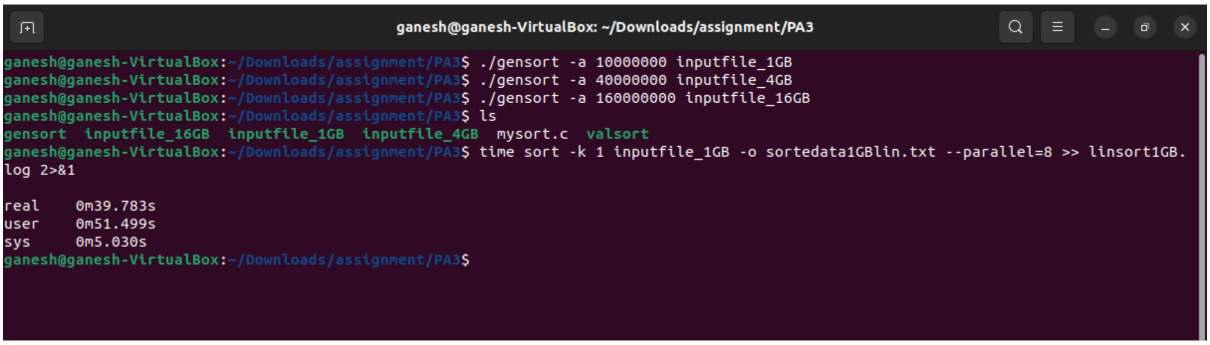
```
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 10000000 input
file_1GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 40000000 input
file_4GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 160000000 inpu
tfile_16GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ 
```



```
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 10000000 input
file_1GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 40000000 input
file_4GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 160000000 inpu
tfile_16GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ls
gensort  inputfile_16GB  inputfile_1GB  inputfile_4GB  mysort.c  valsort
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$
```

Linux sorting for 1GB Data:

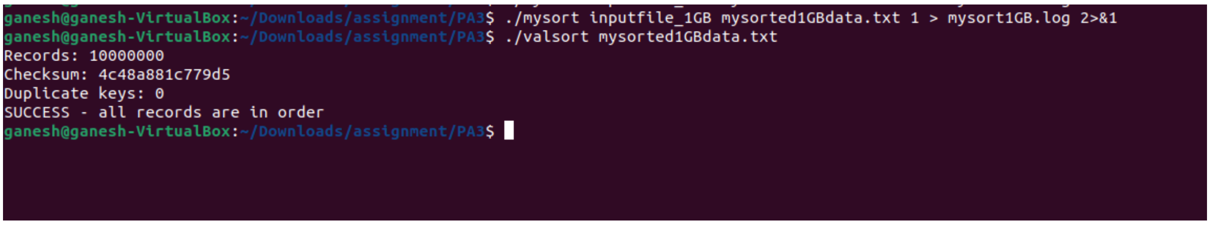
Here for testing purpose we took the thread size as 8



```
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 10000000 inputfile_1GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 40000000 inputfile_4GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./gensort -a 160000000 inputfile_16GB
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ls
gensort  inputfile_16GB  inputfile_1GB  inputfile_4GB  mysort.c  valsort
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ time sort -k 1 inputfile_1GB -o sortedata1GBlin.txt --parallel=8 >> linsort1GB.
log 2>&1

real	0m39.783s
user	0m51.499s
sys	0m5.030s
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$
```

Benchmarking using mysort.c program for 1GB data given thread size as 1 and validating against Valsort



```
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./mysort inputfile_1GB mysorted1GBdata.txt 1 > mysort1GB.log 2>&1
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$ ./valsort mysorted1GBdata.txt
Records: 10000000
Checksum: 4c48a881c779d5
Duplicate keys: 0
SUCCESS - all records are in order
ganesh@ganesh-VirtualBox:~/Downloads/assignment/PA3$
```