External Sort:

Problem Statement

In this programming assignment, we are implementing external sort application in a single node shared memory environment using a multi-threaded approach.

A primarily goal of this assignment is to sort the large than memory size document. We need to work with data generated with the Gensort suite (http://www.ordinal.com/gensort.html) which generates the random data. Total 2 files with size 2GB and 20GB has been generated and placed at path "/input" on Neutron cluster. We need to use above generated files to execute our program. The program can be written in any programming languages such as Java, C, C++. We also need to validate the output generated by our program with valsort program which is a part of gensort suite. We also need to compare our program performance with the standard sorting program such as Linux sort and produce the results.

As we are sorting large document specifically larger than memory, multi-threading approach is must to gain more throughput. The main issue with this type of sorting is requirement of read/write operations to file. To minimize this overhead, I am sorting the block of input file with merge sort and storing the sorted temporary files to disk. Once all records from input file has been read, sorted and stored into one of the temporary files, I am using k-way merging to generate final sorted output file. The input was generated with gensort program (http://www.ordinal.com/gensort.html). Each record is of 100 bytes with starting 10 bytes as key and remaining bytes as value. For this assignment code has been executed for 2GB and 20GB files and results are provided.

Program Methodology

The main bottleneck in the file operations is the disk performance as disk read-write operations throughput is far less than memory. As we are sorting large document specifically larger than memory, we need to minimize the read/write operations. To mitigate these issues, we need to utilize memory as much as we can. As memory is limited and smaller than disk, divide and conquer approach was implemented. The file was divided into number of blocks. Each block has size which can easily fit into memory and sorted in memory before writing it back to disk as temp files. The multi-threading approach was used to sort block of file and store it to disk as temp files. Dividing and working with the block at a time was the main concept for handling external sort.

This approach works with the sorting files in a block size but merging these temporary files and forming final sorted output file is another challenge. The k-way merge approach has been utilized. In this approach 'k'(total number of temp files) are opened and smallest value from this 'k' file has been written to final output file, then 2nd smallest value so and so forth until all 'k' files are empty.

The final output is also validated by valsort program from the Gensort suite to confirm that final sorted output is in order. If there is any problem with order or any duplicates are present in output file, then valsort will report same. Else message "SUCCESS - all records are in order" will be printed. The Linux sort program was also executed, and output file generated from it also validated using valsort. The performance of my program and Linux sort is presented in performance evaluation section.

Runtime Environment settings

The program has been executed on Neutron cluster which has a following environment settings:

- Operating System:
  Ubuntu 16.04.4 LTS, Release: 16.04, Codename: Xenial

- Operating System kernel
  Linux neutron 4.4.0-119-generic #143-Ubuntu SMP Mon Apr 2 16:08:24 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux

- Java Compiler
  javac 1.8.0_162

- Make
  GNU Make 4.1

- JVM specific setting
  Though it wasn’t required but for precaution min and max heap size was provided during execution of 20GB file.

  -Xms64m -Xmx2g

Performance Evaluation

This section presents the results of my program and Linux sort. Below tables shows the results for numbers of threads = 16.

Experiment	Shared Memory (1VM 2GB)	Linux Sort (1VM 2GB)	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)
Compute Time (sec)	81	31	1083	407
Data Read (GB)	4	4	40	40
Data Write (GB)	4	4	40	40
I/O Throughput (MB/sec)	98.7654321	258.0645161	73.86888273	196.5601966

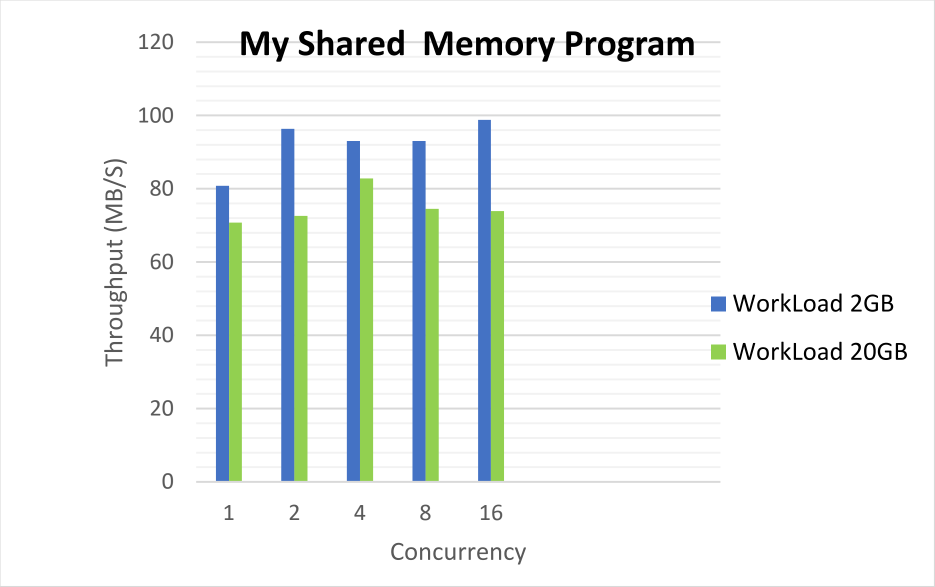
The other experiments mainly with number of threads were also performed. The outcome of such experiments is provided below:

Experiment	**Concurrency**	**Shared Memory (1VM 2GB)**	**Linux Sort (1VM 2GB)**	**Shared Memory (1VM 20GB)**	**Linux Sort (1VM 20GB)**
Compute Time (sec)	1	99	35	1130	376
Data Read (GB)	1	4	4	40	40
Data Write (GB)	1	4	4	40	40
I/O Throughput (MB/sec)	1	80.80808081	228.5714286	70.79646018	212.7659574
Compute Time (sec)	2	83	25	1102	460
Data Read (GB)	2	4	4	40	40
Data Write (GB)	2	4	4	40	40

Experiment	**Concurrency**	**Shared Memory (1VM 2GB)**	**Linux Sort (1VM 2GB)**	**Shared Memory (1VM 20GB)**	**Linux Sort (1VM 20GB)**
I/O Throughput (MB/sec)	2	96.38554217	320	72.59528131	173.9130435
Compute Time (sec)	4	86	36	966	378
Data Read (GB)	4	4	4	40	40
Data Write (GB)	4	4	4	40	40
I/O Throughput (MB/sec)	4	93.02325581	222.2222222	82.81573499	211.6402116
Compute Time (sec)	8	86	37	1073	425
Data Read (GB)	8	4	4	40	40
Data Write (GB)	8	4	4	40	40
I/O Throughput (MB/sec)	8	93.02325581	216.2162162	74.55731594	188.2352941
Compute Time (sec)	16	81	31	1083	407
Data Read (GB)	16	4	4	40	40
Data Write (GB)	16	4	4	40	40
I/O Throughput (MB/sec)	16	98.7654321	258.0645161	73.86888273	196.5601966

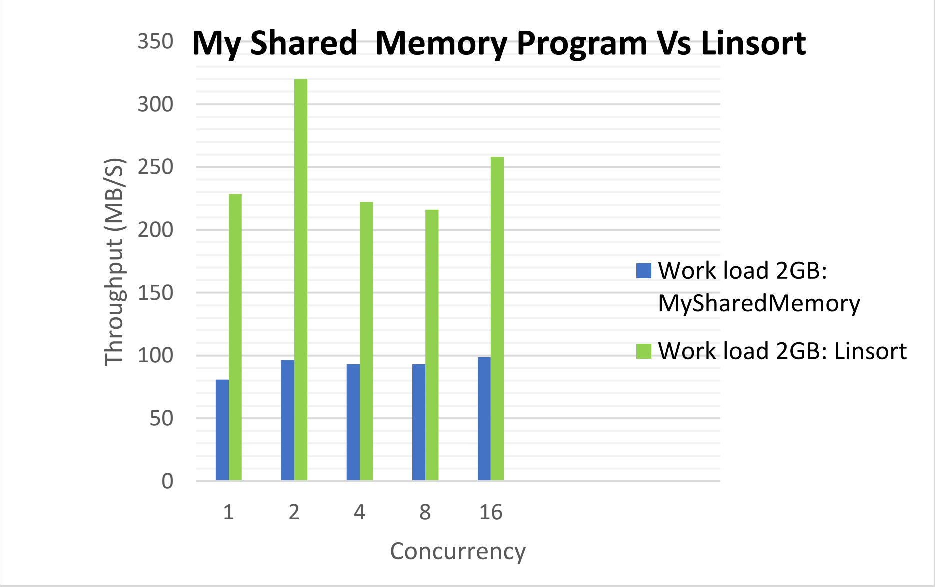
These additional runs have their log and slurm files present in the AdditionalLogFiles and AdditionalSlurmjobs folders respectively in the git submitted.

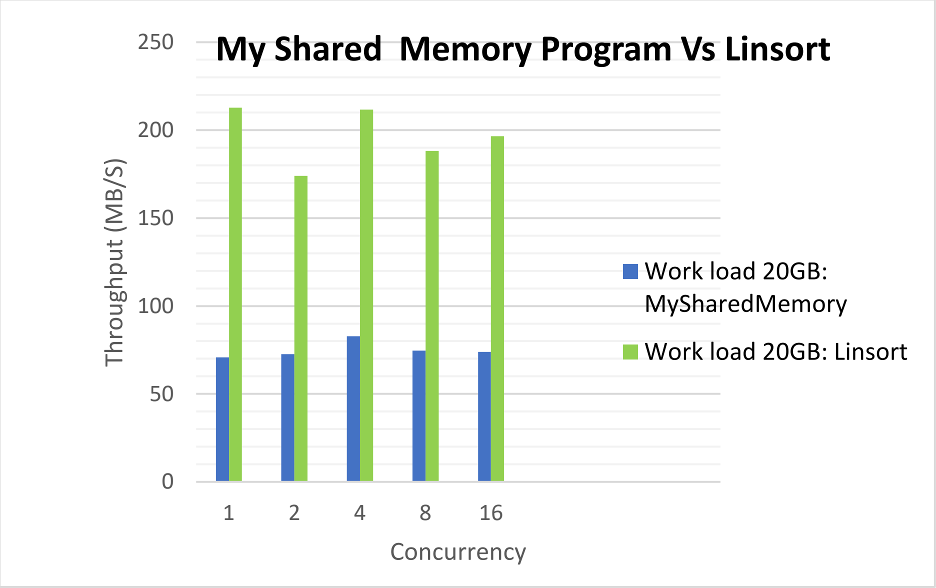
The graphs are also plotted for visual interpretation of above table.



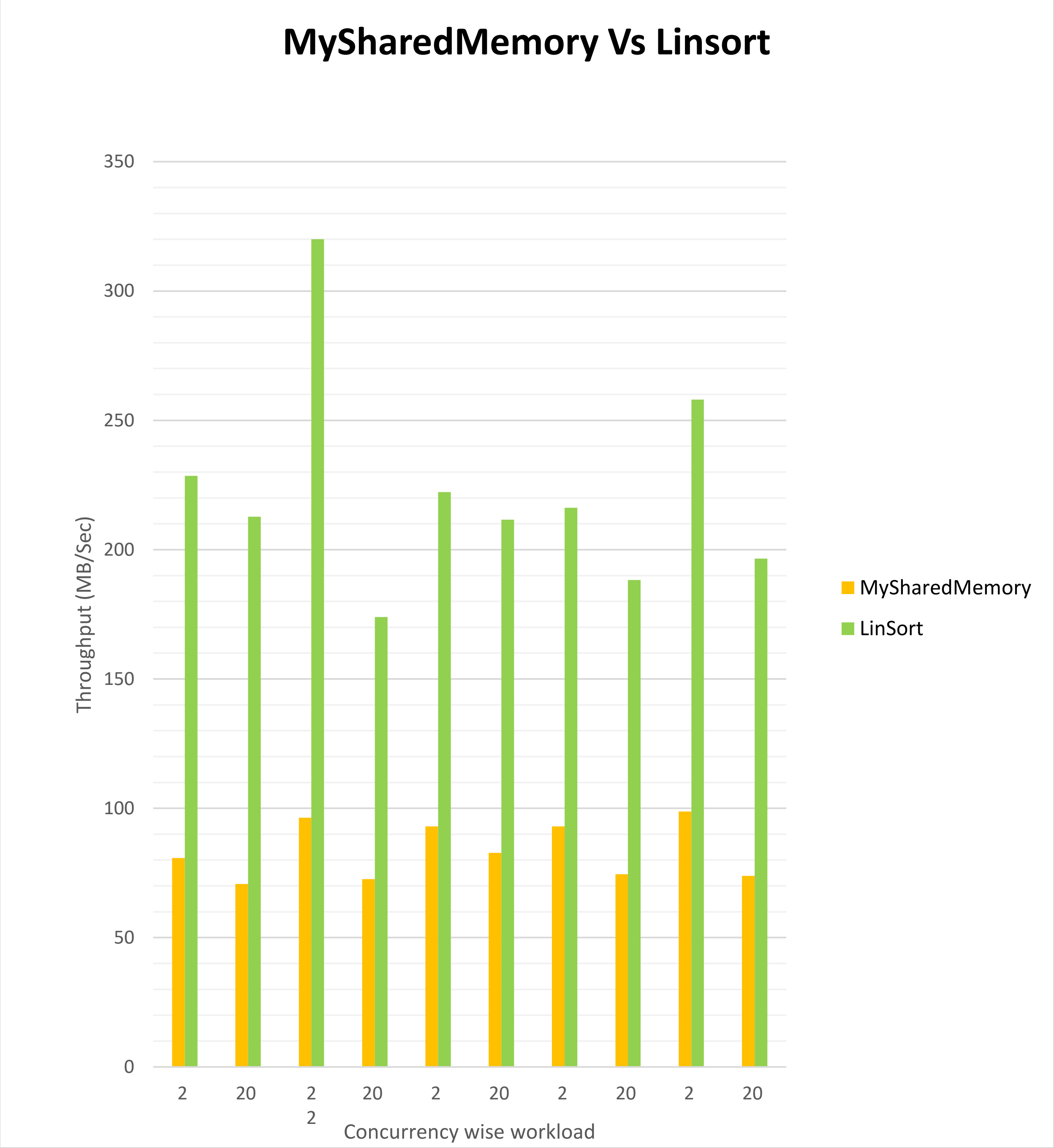
Above graphs indicates the throughput achieved by my program at different concurrency. For work load 2GB throughput increases with number of threads while for 20GB maximum throughput has been achieved when concurrency is 4. The performance of the program decreases with increase in number of threads due to the hardware limitations (no. cores availability, no of disk heads, available memory at time of execution, etc).

Below graphs shows the comparison of my shared memory program performance vs Linux sort program when work load is 2GB. Another graph with workload is 20GB is also plotted.





The difference in performance is because linsort can utilizes the memory and disk both by keeping partial data in both. This will result in more cache hits. Although we don't know actual implementation of linux sort. So, it would difficult to point out exact cause of linsort being better. Also, the limitations stated earlier for my shared memory program can be the reason for having lower performance.



Above graph represents the performance measures for my shared memory program vs linsort with different workload and concurrency level.