Sort on Single Shared Memory node

a. Problem

The assignment focuses on the external sort application to be implemented on a single shared memory node, using multi-threaded approach. The main aim of this assignment is to sort files larger than the memory size, essentially trying to capitalize on sorting capabilities of the program by minimizing read and writes from/to the disk. We must work with randomized data generated using Gensort suite (http://www.ordinal.com/gensort.html) and stored in '/input' folder on the Neutron cluster. There are 2 such datasets of sizes 2GB and 20GB. The program to be designed and implemented can be done using any one of these 3 programming languages, viz., Java, C and C++. Also we have to measure the performance of Linux "sort" command, so that we can compare the performance results from this command against the program we design using the shared memory approach mentioned earlier.

b. Methodology

To tackle the external sorting, an external memory algorithm special care needed to be taken to ensure that there are not many read and writes while sorting the data as disk performance is the biggest bottleneck in this scenario. So, to handle these issues, we needed to identify areas to capitalize on the data handling such it compensates for disk's bottleneck performance, one such area is memory where read and write operations are faster than disk. But as Memory has limited space, Divide and Conquer approach was used to ensure that once data is read from disk, it is sorted in memory before writing it back to disk. This formed one of the core elements for the approach of handling external sorting by dividing the input file into multiple blocks/chunks, sorting them in memory and writing them to intermediate temporary files.

Although this approach helps to alleviate some of the pain points, there remains the task of merging of these temporary files into the final sorted output. K-way merging of the files helps in this scenario, by opening 'k' files and merging the sorted outputs from these files, such that minimum of the sorted outputs from the k file comes first, then 2nd minimum and so on until all the files are empty and all lines written. Thus, we get a final merged output file on the disk, which is then validated by the 'valsort' command from the Gensort suite. If there any mismatched records, then the data records with the first mismatch is output to the terminal along with the total number of mismatched records. In cases of sorted output, the valsort gives the total number of records sorted and success message signifying output is sorted.

c. Runtime Environment settings

The runtime environment settings are as below:

i. *Operating System:*
Ubuntu 16.04.4 LTS, Release: 16.04, Codename: Xenial

ii. *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

iii. *Java Compiler*
javac 1.8.0_162

iv. *Make*
GNU Make 4.1

v. *JVM specific setting – This was used to handle for the large data size especially the 20GB input file, the below is the JVM setting used for the 20GB file so as to give it a minimum of 64MB and a maximum of 2GB heap size, as showcased below.*

-Xms64m -Xmx2g

d. Performance Evaluation

The performance evaluation sections showcase, the results of MySort and Linux Sort programs as shown below. The below values are for execution with 4 threads.

Experiment	Shared Memory (1VM 2GB)	Linux Sort (1VM 2GB)	Shared Memory (1VM 20GB)	Linux Sort (1VM 20GB)
Compute Time (sec)	115	23.604	1092	374.121
Data Read (GB)	4	4	40	40
Data Write (GB)	4	4	40	40
I/O Throughput (MB/sec)	69.565	338.93	73.26	213.83

As seen from the data above the Throughput performance of the Shared Memory model when compared with Linux sort is far lower to the tune of being only 20.52% and 34.26% efficient for 2GB and 20GB loads respectively. Also, from my perspective, the difference in performance might be due to, Linux sort being able to make the most efficient utilization of the Disk and Memory, by getting more hits on caches, having partial data in both Disk and Memory.

Since my design of the Shared memory model uses the Runtime free memory available hence the results achieved will not be constant and will tend to fluctuate based on the load on the compute node. Also other hardware limitations like number of CPU available, no data intensive operations on the disk, etc.

Lastly, if the implementation of Linux sort would have been made available, we could have done a better job in taking care of the smaller niceties of disk and memory accesses to ensure a higher I/O throughput and a better performance.