CS553 – Cloud Computing. Homework 5

Author: Jorge Gonzalez Lopez

CWID: A20474413

Semester: Spring 2021

Problem Statement:

In this assignment, an intelligent code to sort small and large files with different approaches has been created. To that purpose, if the length of the file is smaller than the available memory, an internal merge sort is performed. Otherwise, the file is first split into small chunks which are sorted and saved and then, an eternal merge is ran to create the final sorted output file.

Proposed Solutions:

The internal merge sort works by dividing the input file into two halves, calling itself for the two halves, and, then merges the two sorted halves. This is an efficient 'divide and conquer' sorting algorithm that recursively divides in two halves the input file until the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging back till the complete file is merged. At the same time, each thread will be given an equal load of work to sort so that the algorithm is more efficient.

On the other hand, the external sort has another approach to be able to sort larger than total available memory files. To do so, the algorithm reads the file in small chunks. These chunks are sorted and saved in disk. Afterwards, the first bytes of every chunk are read and sorted so that the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.

The code has been running on a Chamaleon Skylake instance. To that purpose, a ssh connection has been stablished and both, the code and scripts have been uploaded so that the tests could be performed. The Skylake instance has the following resources:

- CPUs: 2
- Threads: 48
- RAM Size: 192 GiB

Therefore, to be able to test the external sort, the memory has been limited as follows:

Java command: The commands of JVM(Java Virtual Machine) have been used. Specifically, `-Xmx8g` which limits the memory to 8 GB.

Linux Sort: The argument `--buffer-size=8G` has been written to also limit the memory.

At the same time, the linux sort has been ran to compare its performance with the one coded. In this case, the argument –parallelism=N has been established to deploy many threads for the file sorting and it has been running with a different number of threads in all the cases to get the one that gave back the best results.

At the same time, to make sure that the data was not cached in the OS memory, the following linux command was ran before each test:

```
sync && echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Finally, as it was asked in the assignment, to monitor the time, memory utilization (GB), and processor utilization (%) as a function of time, the following command has been used:

```
psrecord --interval 1 --log.log --plot graph.png $!
```

where $! stands for the Process ID (PID) of the last process executed in Linux.

Finally, to generate the files, gensort has been used as:

```
./gensort -a -b0 X input
```

where -a stands for generating ascii characters and X is the total number of records (Lines with a length of 100 bytes):

10000000 (1 GB), 40000000 (4 GB), 170000000 (16 GB) and 680000000 (64 GB)

and valsort has been used to verify if the sorting has been ok.

It is important to note that a bash script has been created to run the evaluations that append some additional information to the logs with the commands just mentioned.

Results:

The following table summarizes the results obtained in all the tests:

Experiment	Shared Memory (1GB)	Linux Sort (1 GB)	Shared Memory (4 GB)	Linux Sort (4 GB)	Shared Memory (16 GB)	Linux Sort (16 GB)	Shared Memory (64 GB)	Linux Sort (64 GB)
Number of threads	8	8	8	8	16	32	16	128
Sort Approach	In-memory	In-memory	In-memory	In-memory	External	External	External	External
Sort Algorithm	Merge	Merge	Merge	Merge	Merge	Merge	Merge	Merge
Data Read (GB)	1.00	1.00	4.00	4.00	32.00	16.00	128.00	64.00
Data Write (GB)	1.00	1.00	4.00	4.00	32.00	16.00	128.00	64.00
Sort Time (sec)	19.03	10.02	105.17	61.105	816.37	237.36	2870.74	1234.88
Overall I/O Throughput (MB/sec)	105.10	199.70	76.07	130.92	78.40	134.82	89.18	103.65
Overall CPU Utilization (%)	171.69	278.65	517.68	233.00	700.41	318.00	795.09	278.78
Average Memory Utilization (GB)	3.77	1.28	6.68	4.61	4.64	7.50	4.14	8.05

The total Data Read and Write has been calculated theoretically as the commands iostat and iotop did not give a proper analysis of those values. Therefore, with an in-memory approach the data read and written is the same as the file size, however, in the external case, is the double. This happens because the total data has to be read first to create the chunks and another time

to merge all the chucks into the output file and all the chunks have to be written in disk and then the output file has to be generated again.

Therefore, the Overall I/O Throughput (MB/sec) has been computed as the data read and written divided by the total time.

At the same time, the monitorization of the memory utilization (GB), and processor utilization (%) as a function of time for the 64 GB files are the following:

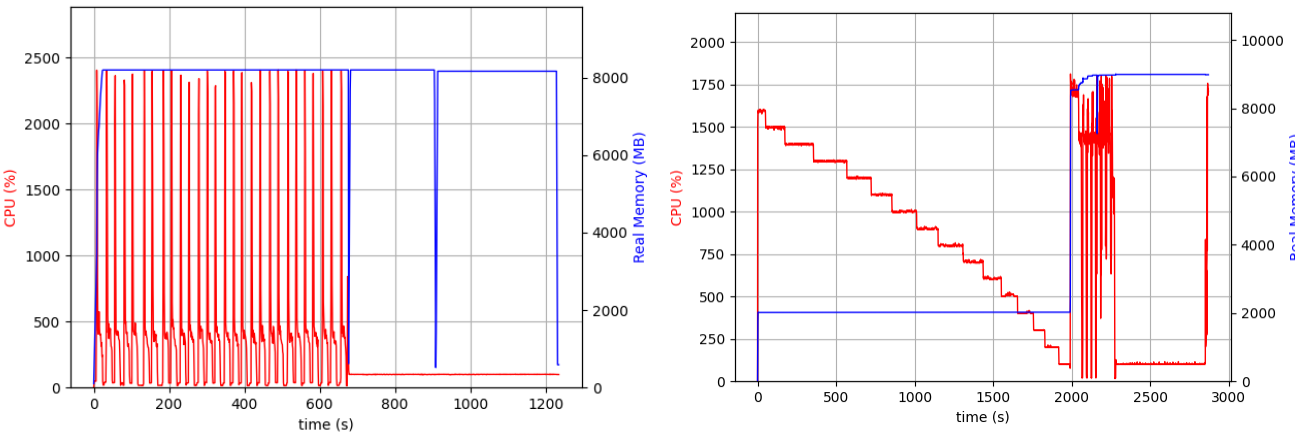

Figure 1: Left: Linux sort. Right: MySort (128 chunks)

As it can be seen in the graphs, the actual sorting is similar in both algorithms. However, with MySort, more than half of the time is spent in creating and saving the file chunks. Therefore, it is less efficient and much slower than the linux sorting. This is also the reason that the average memory utilization is lower that the linux one and the overall CPU utilization greater.