CS553 Homework #5

Sort on Single Shared Memory Node

Instructions:

- *Assigned date: Wednesday April 1st, 2020*
- *Due date: 11:59PM on Thursday April 9th, 2020*
- *Maximum Points: 100%*
- *This homework can be done in groups up to 3 students*
- *Please post your questions to the Piazza forum*
- *Only a softcopy submission is required; it will automatically be collected through GIT after the deadline; email confirmation will be sent to your HAWK email address*
- *Late submission will be penalized at 10% per day; an email to the TA with the subject "CS553: late homework submission" must be sent*

1. Introduction

The goal of this programming assignment is to enable you to gain experience programming with external data sort and multi-threaded programming.

2. Your Assignment

This programming assignment covers the external sort (see https://en.wikipedia.org/wiki/External_sorting) application implemented in a single node shared memory multi-threaded approach.

You can use any Linux system for your development, but you must use the Chameleon testbed [https://www.chameleoncloud.org]; more information about the hardware in this testbed can be found at https://www.chameleoncloud.org/about/hardware-description/, under Standard Cloud Units. Even more details can be found at https://www.chameleoncloud.org/user/discovery/, choose "Compute", then Click the "View" button. You are to use "Compute Skylake" node types; if there are no Skylake nodes available, please use "Compute Haswell" node types, although be aware that the experiments for the larger data sets might take a long time to complete. You are to use the advanced reservation system to reserve 1 bare-metal instance to conduct your experiments. You will need to assign your instance a floating IP address so that you can connect to your instance remotely.

Your sorting application could read a large file and sort it in place (your program should be able to sort larger than memory files, also known as external sort). You must generate your input data by gensort, which you can find more information about at http://www.ordinal.com/gensort.html. You will need four datasets of different sizes: 1GB, 4GB, 16GB, and 64GB.

This assignment will be broken down into several parts, as outlined below:

Shared-Memory External Sort: Implement the Shared-Memory TeraSort application in your favorite language (choose between C, C++, or Java); the next part of this assignment will involve you sorting data in Hadoop (implemented in Java) and Spark (implemented in Scala), which you should keep in mind when you are comparing these implementations. You should make your Shared-Memory TeraSort multi-threaded to take advantage of multiple cores and SSD storage (which also requires multiple concurrent requests to achieve peak performance). You want to control concurrency separately between threads that read/write to/from disk, and threads that sort data once data was loaded in memory. Your sort should be flexible enough to handle different types of storage (where you need different number of threads), and different compute resources (where you need different number of threads based on the number of cores). You may only use the PThread library in C/C++.

In Java, you can only use what is built in to support multi-threading. You must implement your own I/O routines and sorting routines. Since both Haswell and Skylake nodes have more memory than the largest dataset size you have to sort, you must limit the memory usage of your shared memory and Linux sort to 8GB. Note that in-memory sort might be faster than external sort (which has to be used for the 16GB and 64GB datasets. You need to implement a smart enough sort system that will detect the workload size and sort with the best approach possible (in memory for small datasets and external for larger datasets). You should mimic the command line arguments of the Linux sort program for your own shared memory sort benchmark, as much as possible.

Performance: Compare the performance of your shared-memory external sort with that from Linux "sort" (more information at http://man7.org/linux/man-pages/man1/sort.1.html) on a single node with all four datasets. You should vary the number of threads in your shared memory sort and figure out the best number of threads to use for each dataset size. The ideal number of threads might be different for your shared memory sort compared to the Linux sort (see --parallel=N command line argument to sort). Fill in the table below, and then derive new tables or figures (if needed) to explain the results. Your time should be reported in seconds, with an accuracy of milliseconds.

Complete Table 1 outlined below. Perform the experiments outlined above, and complete the following table:

Table 1: Performance evaluation of Single Node TeraSort (using best # of threads for each case)

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								
Sort Approach (e.g. in-memory / external)								
Sort Algorithm (e.g. quicksort / mergesort / etc)								
Data Read (GB)								
Data Write (GB)								
Sort Time (sec)								
Overall I/O Throughput (MB/sec)								
Overall CPU Utilization (%)								
Average Memory Utilization (GB)								

For the 64GB workload, monitor the disk I/O speed (in MB/sec), memory utilization (GB), and processor utilization (%) as a function of time, and generate a plot for the entire experiment. Here is an example of a plot that has cpu utilization and memory utilization (https://i.stack.imgur.com/dmYAB.png), plot a similar looking graph but with the disk I/O data as well as a 3rd line. Do this for both shared memory benchmark (your code) and for the Linux Sort. You might find some online info useful on how to monitor this type of information (https://unix.stackexchange.com/questions/554/how-to-monitor-cpu-memory-usage-of-a-single-process).
After you have both graphs, discuss the differences you see, which might explain the difference in performance you get between the two implementations. Make sure your data is not cached in the OS memory before you run your experiments.

3. What you will submit

The grading will be done according to the rubric below:

- Shared memory sort implementation/scripts: 50 points
- Readme.txt: 5 points
- Performance evaluation, data, explanations, etc: 40 points
- Followed instructions on deliverables: 5 points

The maximum score that will be allowed is 100 points.

You must have working code that compiles and runs on Chameleon assuming Ubuntu Linux 18.04 to receive credit for report/performance. If your code requires non-standard libraries to compile and run, include instructions on what libraries are needed, and how to install them. Furthermore, the code must match the performance presented in the report. A separate (typed) design document (named hw5-report.pdf) of approximately 1-3 pages describing the overall benchmark design, and design tradeoffs considered and made. Add a brief description of the problem, methodology, and runtime environment settings. You are to fill in the table on the previous page. Please explain your results, and explain the difference in performance? Include logs from your application as well as valsort (e.g. standard output) that clearly shows the completion of the sort invocations with clear timing information and experiment details; include separate logs for shared memory sort and Linux sort, for each dataset. Valsort can be found as part of the gensort suite (http://www.ordinal.com/gensort.html), and it is used to validate the sort. As part of your submission you need to upload to your private git repository a build script, the source code for your implementation, benchmark scripts, the report, a readme file, and 6 log files.

A detailed manual describing how the program works (readme.txt). The manual should be able to instruct users other than the developer to run the program step by step. The manual should contain example commands to invoke the benchmark. This should be included as readme.txt in the source code folder.

Here are the naming conventions for the required files:

- Makefile / build.xml (Ant) / pom.xml (Maven)
- MySort.java / mysort.c / MySort.cpp
- Hw5_report.pdf
- readme.txt

- mysort1GB.log
- mysort4GB.log
- mysort16GB.log
- mysort64GB.log
- linsort1GB.log
- linsort4GB.log
- linsort16GB.log
- linsort64GB.log

When you have finished implementing the complete assignment as described above, you should submit your solution to your private git repository. Each program must work correctly and be detailed in-line documented.

Submit code/report through GIT. If you cannot access your repository contact the TAs. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/

Grades for late programs will be lowered 10% per day late.