Homework5 Report

Team-25
Husan-An Weng Lin (A20450355)
Kevin Tchouate Mouofo (A20454613)
Ji Tin Justin Li (A20423037)

Contribute:

- Internal Sorting (Hsuan-An)
- External Sorting (Hsuan-An)
- IO Helper (Justin)
- gensort script (Justin)
- Performance analysis (Kevin)
- Chameleon configuration (Kevin)

The Design of MySort:

1. Overall

In this project, we decided to divide our MySort Program into three parts internal sort, external sort and IO Helper. The internal sort will take care of every in-memory sorting. The external sort will be used to merge the files generated by the internal sort. Since we need to access the disk frequently, we designed classes to handle all IO operations.

2. Internal Sort (Hsuan-An)

For the internal sort, I choose heap sort as the sorting algorithm. Before I choose the heap sort, I actually want to use merge sort since that is stable and the performance is pretty good too. However, normal merge sort needs an extra array to do the sorting which means that we need more memory for internal sort. If the memory size is 8GB we probably can only sort 4GB data in once. Therefore, I choose the heap sort which also has good time complexity but the space complexity is O(1). Although it is not stable(the order of the same record will be changed), I think for our purpose, it won't matter.

3. External Sort (Hsuan-An)

I think this part is the most difficult part. To merge the sorted file, we actually have a lot of options so we discussed a lot and finally decided to use K-way merge which we think is more efficient. For the K-way merge, we are using min-heap data structure to merge multiple files at the same time. Also, to deal with the IO bottleneck, I design a kind of buffer mechanism. I believe that rather than read the records from the sorted files one by one, we can actually read many records to the memory so that we can save a lot of time on IO. There are two different types of buffer when we do the external sorting. One buffer is for the output which will store the merge data pop out from the min-heap tree. When the buffer is full, it will write the data to the disk and then flush the buffer so that the output buffer can store more data. The other type of buffer is the input buffer which is for those sorted files that need to be merged. It will read the data from those files by the chunk size and the buffer will feed records to the min-heap tree to do the merge. When the buffer reaches the and of it, it will read another chunk of data from the same file to the buffer. For now, I dynamically set the output buffer size to half of the memory we can use and the input buffers will evenly take the rest of memory. For example, if we have 8GB for the memory and we have 4 files to merge, the output buffer will be 8/2 = 4GB and the input buffer will be 4/4 = 1GB.

The possible improvement of this part I think is on the way we assign the buffer size. For easy to implement, I assign the memory to the buffers by a very simple formula which I believe is not ideal in every situation. For better performance, we should design a better way to utilize our memory usage.

4. IO Helper (Justin)

Overview of classes: IO Helper handles disk access and loading of the workloads into memory. It exposes the workload file to the program with simple access functions for easier file manipulation. Two classes(IO_Helper and Buffered_IO_Helper) are created where both classes provide (almost) identical functional interfaces. Buffered_IO_Helper acts as a wrapper class of IO_Helper which also manages multithreaded chunks buffering when reading in workloads.

Implementation details: The header file “io.h” contains all the declaration of the two classes. The function definitions are split into two separate files, namely “io.cpp” and “buffered_io.cpp”. Each IO_Helper, buffered or not buffered, handles the disk IO of one file.

IO_Helper: To use the single threaded class IO_Helper, the constructor takes in two arguments, workload filename and the desired chunk size. The access function readChunk() uses <fstream> to read a section from the file. It uses malloc to allocate a chunk of memory, and sequentially and loads the chunk into memory as a string array. IO_Helper itself keeps track of the file offset with a member variable, and isChunkAvailable() lets the user function know if the file has reached EOF. IO_Helper also supports write to file via the access function writeChunk(), which writes a string array to the specified file.

Buffered_IO_Helper: The constructor of Buffere_IO_Helper takes in an extra argument which specifies the size of the buffer available to be used by this object. When constructed, the object creates a c++ std::queue<std::string*> to hold the buffered chunks. An additional start_thread() function starts up a pthread for this object, which continuously reads in chunks(with IO_Helper access functions) from the workload file until it reaches EOF. Concurrency control is done with POSIX semaphores. The readChunk() and writeChunk() access functions exposed to the other functions behave exactly the same as the internally wrapped object, IO_Helper.

Further Improvement: Currently Buffered_IO_Helper only multithreads readChunk(), while writeChunk() still uses the main thread to carry out operations. It is possible to multithread writeChunk() similarly to what was done in readChunk() to further improve performance.

5. Gensort script (Justin)

The script is located in src/scripts/createData.sh. The script to generate our workload files is extremely simplistic since only 4 different workload sizes are needed. Once called, it creates a data directory and calls to gensort, which creates 4 files, namely: *gs.out.1g* | *gs.out.4g* | *gs.out.16g* | *gs.out.64g*, of size 1GB, 4GB, 16GB and 64GB.

6. Resource usage (Kevin)

To monitor the resource usage of our program for the 64GB external sort and come up with a graph, I have created resource.cpp. Resource.cpp is a piece of code that creates a file with the information of our external sort: memory utilization, and processor utilization. The resource code was called by the external sort during its execution for the 64GB file. We have decided to compute the disk I/O separately because it was easy for us. After all the files were generated, we draw the graph with a program in R.
The resource usage of the 64GB linux sort was obtained by running the psrecord command. Afterwards we compared our results.

7. Linux sort (Kevin)

To monitor the resource usage of the linux sort and mysort, we have used the htop command with a different set of screens. So, the linux sort was running on a screen and on another screen the htop command was showing the diverse information:

- NLWP : The number of threads in the process.
- IO_RATE (IO): The overall throughput.
- PERCENT_CPU (CPU%): The percentage of the CPU time that the process is currently using.
- M_SIZE (VIRT): Size in memory of the total program size.

For the percentage of the CPU used (CPU %), and the overall throughput (IO), we measured some values and gave the mean and the standard error as shown in our table. Unfortunately, we did not have enough time to run our code for resources utilization graph.

8. Discussion

Our mySort program can actually sort the data including large workloads by using an external sorting mechanism. However, we might not utilize the memory usage pretty well. First of all, for the internal sorting, if the memory is larger than the workload, we should use the extra memory to improve the performance. But due to the

time constraint we don’t have time to utilize this part. Secondly, after running the monitor tool, it turns out that we might use more memory than we expected. We think it is probably because we didn’t do the garbage collection well, so our program keeps using new memory during the process without freeing unused one. Thirdly, our threading mechanism seems like it doesn't work as we expected. We thought it should use more threads in larger workloads but it doesn’t actually improve the performance a lot when we do the large workload. We might need to rethink the way we multithread.