CS3103 Project Report: Parallel TopK

Group 30

SEW Kin Hang(56614778, khsew2-c@my.cityu.edu.hk)

LOK Chun Pan(54396020, cplok2-c@my.cityu.edu.hk)

Introduction

In our project, we first start with a single-threaded version to ensure the correctness of the program; then, we begin to implement multithreaded file reading, parsing, and sorting to boost the time further.

Through the development process, we figured out that the project can be divided into two parts: (1) read and parse a single large file efficiently, and (2).read and parse multiple small files efficiently.

In our final design, we have developed two modes in the program. The first mode is suitable for a single file, and the second mode is ideal for multiple small files. The program will determine its mode during runtime by counting the number of files in the test case directory.

File Reading and Parsing

Overall Design

The program will read file by file and line by line, then store each entry into the counter array.

Special Design 1: Producer Consumer Model for Multi-File Reading

This design implements a single buffer producer-consumer model for file reading. We adopted the model to read multiple files with multiple threads concurrently. In the design, one producer produces file paths as items to the buffer; then, numerous consumers will fetch file paths and

perform reading and parsing. Although the line reading is still performed one by one, it is already highly efficient for test case 5 because there are a lot of small files inside the test case.

Special Design 2: Producer Consumer Model for File Parsing

This design implements a producer-consumer model for file parsing. The reader reads the line to the buffer, and then the parser continues his work. An interesting design for this is that we reversed the actor of producer and consumer. The parser is the producer, and the reader is the consumer. The item will be the producer itself's identifier. This design is because concurrent storing can be performed without the need for a mutex lock.

For example, if we now let readers be the producer and parsers be the consumer, the produced item will be characters. After that, the parsers have to work in the critical zone to copy the characters or parse strings directly, which will block other parsers.

However, if we swap the roles of producers and consumers, the file parser will put itself into the buffer when it is free to work. File readers will get one available parser and read the file to the parser buffer, which is more efficient that this action doesn't have to perform in the critical zone.

Using design 2 is efficient for test case 4 but is less efficient for test case 5 than using design 1.

Special Design 3: Different modes for Different Test Cases

As design 1 and design 2 have different advantages when handling different files, the final version combines design 1 and design 2 to form a single program with two modes. The program will run the corresponding mode according to the number of files in the test case.

Abandoned Design

We tried to read the file with fgetc instead of fgets, but it terms out to be slower. We believe this may due to the overhead of function calls.

TopK Sorting Algorithm

Overall Design

We implemented a multi-threaded heap sort; the topK operation is separated into two threads and then merged with two sorted topK results. We combined merge sort and heapsort as a recursive function. The function will perform heapsort when the number of elements in the recursive part is smaller than a certain threshold. Otherwise, the function will perform merge sort.

However, we found that the difference between multi-threaded heap sort and single-threaded heap sort is small, only boosted around 0.1seconds. We believe the sorting process is not the bottleneck of the project because the value of n and k is small.

Memory Usage

Overall Design

There is a 50MB memory limit; therefore, it is impossible to store all the entries in the program or count all the entries in seconds. To reduce memory usage, since the output only requires the frequency in hours, we only need to create a counter array storing the frequency of entries in each hour. The total hours in the data range is 2^25, so the total hours is 2^25/60/60=9321. Therefore, the counter array can fit the memory limit after aggregation.

During the development process, we met the problem of insufficient memory. When we test the program with the script, although it doesn't exceed the memory limit of 50MB, it works slowly and can't pass the time limit. We confirmed this is a memory problem because the program runs perfectly outside the script with 'make test.' Therefore, we had to spend extra time modifying the design further.

Number of Threads

The number of threads shouldn't be too large because we have met a memory problem using a large number of threads. After testing, we found that for small numbers such as 1 and 2 it will lower the speed. However, for larger numbers, the speed doesn't variate a lot.

Other Techniques

Usage of Macro

In C language, we can use macros to allow the compiler to perform some preprocessing. We adopted macro instead of const variables in some places to declare some constant values to speed up the program. We also adopted macro as a compare function so that there is no function overhead.

Fgets() vs Fgets_unlocked()

Originally, we used fgets() to get lines in inputs. However, after searching for a faster input method, we found fgets_unlocked(), a no-lock version of fgets(). It is a thread-unsafe operation when two threads read the same file together, but since our design only has one thread for each file, we don't have to implement a lock for fgets_unlocked.
After changing fgets() to fgets_unlocked(), the code runs much faster for test case 4.

Custom string to long function

Originally, we used strtol() to convert the required timestamp to long to get lines in inputs. To further optimize, we implemented a custom string to long function. As we already know the length of the timestamp and we know it is not a negative number, we can save some conditions for special cases. Therefore, the program runs faster using a customer string to long function.

Conclusion

After implementing different solutions, we believe the most crucial part of this project is to handle file reading and parsing correctly. We have both test case 4, which illustrates a single large file, and test case 5, which shows the situation of having many files. If we can handle these two cases efficiently, we can obtain a fast result. It is less important for other factors such as the topK sorting process because there are not many counters for us to sort after aggregating the memory.

Although having two modes in our program to handle testcase 4 and 5 seems to be tricky, it is not a poor design because in reality, it is difficult to balance between two situations. Owing to the time limit, having two separated algorithms for two situations is a desirable way to achieve a better result instead of spending extra time generalizing the algorithm.

For future improvement, if I have more time, I would like to try mmap because I think putting the records into the memory may have better random access performance so that parallel reading for different parts in the same file is possible. However, I am not sure it will boost the result to what extent. It is because, in the project, we have to read all the entries, also we have a limited memory space of 50MB, which mmap may not perform at its best as mmap benefits random access and large memory space.

The second improvement that can be done is to enlarge the buffer size in the producer consumer model. In our design, the size of the buffer in the first producer consumer model is 1. If time allows, larger buffer size can be implemented to improve the performance.