Project: Parallel TopK

Submission Due: Friday, April 15, 2022 at **8 PM HKT**. Late submissions will be penalized (10 marks deducted if late for <=12 hours, 20 marks deducted if late for 12-24 hours, and so on).

This is a group project. Each group has 2~3 members (the grading standard are the same for all groups, regardless of the group sizes).

I. Project Instructions

Overview

In the programming question of the assignment, you have implemented a basic version of the TopK algorithm, for which you are supposed to use heap sort to implement a **single-threaded** TopK algorithm. In this project, you are expected to design and implement a **multi-threaded** TopK algorithm and apply it to a series of external files.

Trace analysis is an important component in our daily development, especially for the modern HTTP server system. So, we simplify the HTTP log as the description in Section II and perform a typical operation: Finding the most frequently accessed sites.

There are three specific objectives in this project:

- To learn how to perform a multi-threaded reading on different workloads.
- To learn how to arrange the parallelism between file reading and file block parsing.
- Understanding how to reduce memory usage by aggregating information.

Each group is required to submit the following:

1. **Source code file(s).**
2. **The compiled executable file (complied on the gateway Linux Server).** The executable file should be named as "XX-test.o", where XX represents your group number. For example, if you are in group #12, then your submitted executable should be named as "12-test.o".
3. **Project report**. The project report should concisely describe the design, implementation, and experimental results. **Please list the names, student numbers, and emails of all group members in the project report and describe each team member's contribution.**

Project Organization

Each group should do the following pieces to complete the project. Each piece is explained below:

- **Design** **30 points**
- **Implementation** **30 points**
- **Evaluation** **40 points**

Design

The design space of building an effective TopK algorithm is large. A practical PTopK (Parallel-TopK) that achieves a better performance will require you to address (at least) the following issues (see part *II. Project Description* for more details):

- How to parallelize the TopK sorting algorithm.
- How to efficiently sort the input data of HTTP access log by using PTopK concurrently.
- How to determine the number of threads to create.
- How to access the input file efficiently.

In your project report, please describe the main mechanisms proposed to parallelize the TopK algorithm, list and explain all the functions used in this project, summarize and analyze the results. You could also discuss any difficulties encountered and the lessons learned in solving them.

Your solution should comply with the following requirements:

- This is a TopK algorithm. Even in the worst case, the Time complexity should be O(K*N), so we set the time limitation for 5 secs, which is long enough for a single-threaded heap sorting algorithm to finish the job. This means if you are running a full sorting, your program will face the time-out exception and won't get the score in the target test case.

- We also set the 50 MB limitation of memory usage, if your program uses more than 50 MB memory, your program will be killed at once, and surely your output can't match the expected ones, therefore you won't get the score in that test case. Although we mentioned the possible value range, it's not possible for you to simply use the bulk sort to do the comparing part.

Implementation

Your code should be nicely formatted with sufficient comments. Each function should be preceded by a header comment that describes what the function does. The code should be easy to read, properly indented, employ good naming standards and structure, and correctly implement the design. Your code should match your design.

In case your code cannot be compiled or does not give the correct result, you may still get part of the implementation scores according to the quality and degree of correctness of the codes.

Evaluation

We provide 5 test cases for you to test your code (note: we will also use many other test cases when grading your submission; these 5 test cases are just to help you to evaluate your own solution). Before submitting your work, please run your PTopK on the test cases and check whether your PTopK is sorting the data of input files correctly. A time limit (i.e., the maximal time allowed to finish the sorting) is set for each test case, and if this limit is exceeded, your test will fail. For each test case, your grade will be calculated as follows:

- ***Correctness***. Your code will first be evaluated by correctness, ensuring that it sorts the data of input files correctly. You will receive full points if your solution passes the correctness tests performed by the test script. You will receive **zero points** for this test case if your code is buggy or become too time costly. (e.g., if you pass all the test cases, you can get 100% in your implementation part, but if you pass only 3 test cases, you can only get 60% of your implementation grade.)
- ***Performance***. If you pass the correctness tests, your code will be tested for performance; the Test script will record the running time of your program for performance evaluation. Shorter time/higher performance will be granted bonus scores (will be explained in the latter part of this document).

In your project report, summarize and analyze the results. You can also compare your solution with the provided baseline implementation.

Tips*: Keep a log of work you have done. You may wish to list optimizations you tried, what failed, etc. Keeping a good log will make it easier to put together your final writeup.*

Other Remarks:

Language/Platform

You are recommended to use ANSI standard C for this project. You are also allowed to use C++ (but we don't provide the makefile for C++, so you have to take care of it by yourself). Please also notice the bonus calculation policy regarding using C or C++, which will be explained in the latter part of this document.

You can develop your program on Linux (recommended), MacOS, or Windows using Cygwin. Since grading of this project will be done using a gateway Linux server, the teams that choose to develop their code on any other machine must run and test their programs on the gateway Linux server before submission (it's your own responsibility to make sure your code can run properly on the gateway Linux server). There will be no points for programs that do not compile and run on the gateway Linux server, even if they can run well somewhere else.

Bonus

20 bonus scores will be granted to each team whose program's performance is in the top 20% (rounded down) among all teams. To encourage healthy competition and motivate you to improve the performance of your program, we'll provide a scoreboard that shows scores (reflecting the performance) for each group. The latest scores with ranking are displayed on the scoreboard and updated every day. More details about the scoreboard will be posted in Canvas later.

The total coursework score is capped by 40% of the entire course grade with the bonus scores, i.e., the bonus scores do not transfer to your final exam.

Please note that although we recommend you to use C to finish the project, some teams may choose to use C++. The competition will be separated into two streams, one for all C teams and one for all C++ teams. For example, if in total 6 teams choose to use C++, then 1 team (20% and rounded down) in the C++ stream will be grant the bonus score. If in total there are less than 5 teams choose to use C++, then no team in the C++ stream will be grant the bonus score.

Handing In

Each group can have a maximum of three members. All students are required to join one project group in Canvas: "People" section > "Groups" tab > "Project Groups" Group > Project Group 01– Project Group 70.

Self sign-up is enabled for these groups. Instead of all students having to submit a solution to the project, Canvas allows one person from each group to submit on behalf of their team. If you work with partner(s), both you and your partner(s) will receive the same grade for the project. Please make sure to write down all group member's particulars (name, SID, and email) in the project report.

When you're ready to hand in your solution, go to the course site in Canvas, choose the "Assignments" section > "Project" group > "Project" item > "Submit Assignment" button, and upload your files, including the following:

1) A project report in PDF format, which concisely describes your design, implementation, and experimental results;
2) The source code file(s);
3) The compiled executable file (compiled on gateway Linux server). The executable file should be named as "XX-test.o", where XX represents your group number. For example, if you are in group #12, then your submitted executable should be named as "12-test.o".

Academic Honesty

All work must be developed by each group independently. Please write your own code. **All submitted source code will be scanned by anti-plagiarism software**. We will both test your code with the test cases (including the 5 test cases provided to you and many other test cases), and **carefully check your source code**. If your submitted code does not work, please indicate it in the report clearly.

Questions?

If you have questions, please first post them on Canvas so others can also get the benefit of the TA's answer. Please do not post your source code that will reveal your solution to others or allow others to abuse it. If the posts on Canvas do not resolve your issue, please contact the Tutors, Mr. YU Jianghuan <jinghuan.yu-c@my.cityu.edu.hk> and Ms. REN Tianyu <tianyuren2-c@my.cityu.edu.hk>.

Acknowledgments

This project is taken and modified from the OSTEP book written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin. This free OS book is available at http://www.ostep.org. Automated testing scripts are from Kyle C. Hale at the Illinois Institute of Technology.

Disclaimer

The information in this document is subject to change **with** notice via Canvas. Be sure to download the latest version from Canvas.

II. Project Description

For this project, you will implement a parallel version of topk using multiple threads. First, recall how the topk algorithm works by reading the description in Assignment 1 Part II, where we recommend that you use heap sort to implement a single-threaded TopK sort.

1. Input description:

In this project, we provide the simplified log in the following format:

1. Each line of the log represents an access record, and we call it a **log entry**.
2. All lines are separated from others by the character **\n**
3. Each entry consists of a timestamp and a random length string, and they are separated by the comma. The timestamp is a 10-bytes integer, starts from 1645491600 and end at 1679046032 (Tuesday, February 22, 2022 1:00:00 AM to Friday, March 17, 2023 9:40:32 AM)

```
1645491600, somerandomword
```

4. To simplify your work, all files are start and end with completed lines, which means **you don't need to worry** about the missing rows like this:

```
91600, somemissing
```

2. Output description:

After you have already searched through all the input files, you should write out the most frequently accessed **hour** with the access time of it, separated by \t, and each line end up with an \n, (DO NOT miss the first sentence.):

```
Top K frequently accessed hour:
Fri Mar 17 16:00:00 2023        3600
Fri Mar 17 15:00:00 2023        3600
Fri Mar 17 14:00:00 2023        3600
Fri Mar 17 13:00:00 2023        3600
Fri Mar 17 12:00:00 2023        3600
```

Please notice that, when the frequency is the same, you should order the entry by the time (from larger to smaller).

Still, just like the coding part in Assignment 1, we will provide you an example program for you to check your answer.

3. Useful hints:

Doing so effectively and with high performance will require you to address (at least) the following issues:

- **How to parallelize the Topk sorting process.** The main challenge of this project is to parallelize the file accessing tasks and Topk sorting process. You are required to think about whether the sorting process can be separated into several sub-processes, what sub-processes can be done in parallel, and what sub-

processes must be done serially by a single thread. Then, you are required to design your parallel sorting as appropriate.

For example, does it possible to sort several small sub-files using multiple threads instead of sorting a large file using only one thread? If it's possible, how to divide the large file? How to sort those small sub-files using multiple threads? How to merge sorted results of several small sub-files? One interesting issue is, using different threads to read separated parts of the file may not achieve better performance, disks on the different environments may have different parallelism and bandwidth limitation. We would recommend you think about the producer-consumer model, for it will make the threads focus on their own job and gain better performance.

- **How to parse the file with a specific format.** Parsing the string could be a very challenging part of this project. This part is highly related to the page frame and segmentation method in memory management. You can either read the file in specific format, this will hurt the read performance, but OS will take over the formatting through highly optimized macros. Or you can read the file with a certain length and do the parsing by yourself, which provides a much higher read throughput. The pipeline design will influence your program efficiency, and, to provide enough thinking depth, it's also highly related to the CPU branch prediction and sorting algorithm deterioration problems. In some cases, the target file is ordered already, but how to access it? How to accelerate it by this feature? Would it bring harm to your sorting method? These are a worth concern.

- **How to determine the number of threads to create.** On Linux, the determination of the number of threads may refer to some interfaces like `get_nprocs()` and `get_nprocs_conf()`; You are suggested to read the man pages for more details. Then, you are required to create an appropriate number of threads to match the number of CPUs available on whichever system your program is running.

- **How to decrease the memory usage by aggregating the entries.** So, you will learn about the Pagetable in the chapter on memory management. In this project, you will be asked to record the results that have 2^25 possible values, which will take more than 600 MB memory to store. However, we set the hard limitation of memory usage of 50MB, and please think about how to save the space for it. Could it be possible to create a dynamic array that allocates memory only when it's needed? Or something you can refer to the multi-leveled page table design.

III. Project Guidelines

Getting Started

The project is to be done on the CSLab SSH gateway server, to which you should already be able to log in. As before, follow the same copy procedure as you did in the previous tutorials to get the project files (code and test files). They are available in */public/cs3103/project/* on the gateway server. `project.zip` contains the following files/directories:

```
/project
├── makefile
├── test.example              <- The sample solution (executable file).
├── ptopk.c             <- Modify and hand in this file.
├── multi_threaded_read_and_parse.c
├── multi_threaded_read.c
├── multi_threaded_segment_read_and_parse.c
├── multi_threaded_segment_read.c
├── single_threaded_read_and_parse.c
├── single_threaded_read.c
├── case1
│   └──input0
├── case2
│   └──input0
├── case3
│   └──input0
├── case4
│   └──input0
└── case5
    ├── inputaa
    ├── ...
    └── inputaaz
```

Start by copying the provided files to a directory in which you plan to do your work. For example, copy */public/cs3103/project/project.zip* to your home directory and extract the files from the ZIP file with the `unzip` command.

```
$ cd ~
$ cp /public/cs3103/project/project.zip .
$ unzip project.zip
$ cd project
$ ls
```

There are many example cases for you to refer to so that you can learn how to read the file in different ways and how to check and open a directory. A sample `test.example` is also provided (we only provide a single

executable file without source code). It's a single-threaded program, so any implementation which is slower than it will be determined as timeout (but don't worry, we will leave some space for the variance). **Also, any implementation that uses over 50MB will be instantly killed by the judging script**, so place think about how to save the memory overhead.

4. Writing your `ptopk` program

The ptopk.c is the file that you will be handing in and is the only file you should modify. Write your code from scratch or simply borrow the data structures defined in the example codes to implement this parallel version of topk algorithm. Again, please notice that the worst case of topk should be O(K*N), so efficiency is another thing you should always care about.

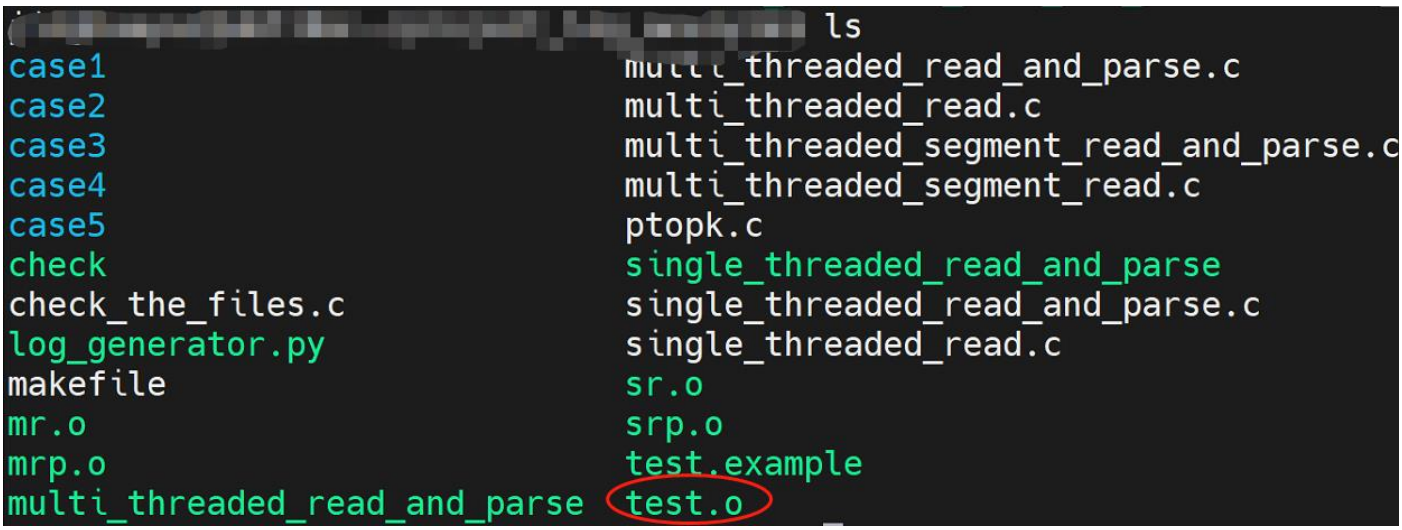
5. Building your program

A simple makefile that describes how to compile `ptopk` is provided for you.

To compile your pzip.c and to generate the executable file, use the `make` command within the directory that contains your project. It will display the command used to compile the `ptopk`.

```
$ make
cc ptopk.c -o test.o -l pthread
cc single_threaded_read.c -o sr.o -l pthread
cc multi_threaded_read.c -o mr.o -l pthread
cc single_threaded_read_and_parse.c -o srp.o -l pthread
cc multi_threaded_read_and_parse.c -o mrp.o -l pthread
```

If everything goes well, there would an executable file `test.o` in it:



If you make some changes in pzip.c later, you should re-compile the project by running `make` command again.

To remove any files generated by the last `make`, use the `make clean` command.

```
$ make clean
rm -f *.o *.out
```

6. Testing your C program

We also provide 10 test cases for you to test your code. You can find them in the directory tests/tests-pzip/. The makefile could also trigger automated testing scripts, type:

```
$ make test
echo "here are the reference output"
here are the reference output
./test.o case1/ 1645491600 5
./test.o case2/ 1645491600 5
./test.o case3/ 1645491600 5
./test.o case4/ 1645491600 5
./test.o case5/ 1645491600 5
echo "here are the reference output"
here are the reference output
./test.example case1/ 1645491600 5
Top K frequently accessed hour:
Sun Feb 26 20:00:00 2023       6
Mon Jan  2 02:00:00 2023       6
Thu Nov  3 19:00:00 2022       6
Thu Jul 21 16:00:00 2022       6
Sat Feb 25 07:00:00 2023       5
./test.example case2/ 1645491600 5
Top K frequently accessed hour:
Wed Dec 14 02:00:00 2022       7
Thu Dec  1 01:00:00 2022       7
Sun Feb 12 03:00:00 2023       6
Thu Dec  1 03:00:00 2022       6
Sat Nov  5 05:00:00 2022       6
./test.example case3/ 1645491600 5
Top K frequently accessed hour:
Fri Feb 10 01:00:00 2023       6
Fri Feb  3 00:00:00 2023       6
Sat Dec 24 01:00:00 2022       6
Sat Feb 18 07:00:00 2023       5
Mon Feb  6 13:00:00 2023       5
./test.example case4/ 1645491600 5
Top K frequently accessed hour:
Fri Mar 17 16:00:00 2023       3600
Fri Mar 17 15:00:00 2023       3600
Fri Mar 17 14:00:00 2023       3600
Fri Mar 17 13:00:00 2023       3600
```

```
Fri Mar 17 12:00:00 2023       3600
./test.example case5/ 1645491600 5
Top K frequently accessed hour:
Thu Nov 10 10:00:00 2022       5
Tue Mar 14 16:00:00 2023       4
Thu Dec 22 04:00:00 2022       4
Tue Dec 20 10:00:00 2022       4
Fri Nov 25 01:00:00 2022       4
```

We also provide the test.py for you to check about whether your program has been time out or not. Also, it will compare the output of your test.o and reference output and instantly kill the program that exceeds the memory limitation.

Below is a brief description of each test case:

- Test case 1-3: These are the test cases for you to get familiar with the basic file handling and external sorting methods. Each file consists of 10000 lines of a log entry. Test case 1 is unordered, test case 2 and test case 3 are in the opposite order. (Mostly ordered, we add some noise data in the test case, so the approach that simply reads the head and tail can be abandoned for now).
- Test case 4：This is the test case that iterates through every single log entry but consists of only one file that stores all the entries.
- Test case 5：This test case generates 2^24 different entries and shuffles them into different files.

PS: Please notice that we will also use many other test cases to test and evaluate your programs and carefully check your source code whenever needed.