CS3103 Project: Parallel TopK

Semester B 2021-22

KOZHIN Assan

SALTER Glenn

TOROMANOVIC Jovan

Contents

Algorithm Design

Task can be separated into several parts:

1. Data aggregation (including data structures to store data)
2. File read and parsing
3. TopK algorithm

Data Aggregation

Problem statement

The problem area can be simplified by using some optimizations on the input data. Each log entry consists of a timestamp, and some random string, and we are only concerned about the timestamp. Each timestamp consists of 10 digits, e.g. 1645491600. The values of each timestamp ranges from 1645491600 to 1679046032. Hence, there would be 33554432 possible timestamp values for each entry log. If we were to use a counter array to store the number of occurrences of each timestamp, we would require 2^25 Bytes or 32 MB, assuming each element takes up 1 Byte.

Our solution

Since we are only limited to 50MB of memory usage for the program, we cannot afford spending 32MB for the counter array only. But timestamps only need to be priced to the hour, so we can ignore the minutes, and seconds of the timestamp. Thus, storing a smaller range of values and saving memory.

This range can be stored in a simple array, and is treated as a hash-map with operations discussed above as a hash function.

$$h(x) = (x\text{-min\_timestamp}) \ / \ 3600$$

*Note that integer division was used here.

From the given formula above, we aggregated the data by subtracting the starting date from the timestamp in each entry log, and then dividing this value by 3600, this value would range from 0 to 33554432. Then we divide this value by 3600, which reduces the number of digits used in a timestamp by at least 3. In practice, this gives us a range of pseudo timestamp values ranging from 0 to 9321, since there are only 9321 hours between the given timestamps in the problem

statement. Therefore, the counter array only needs about 40KB, which is quite a small data structure compared to the initial array in the problem statement.

File Reading and Parsing

We identified that file reading and parsing is the bottleneck of the algorithm, and most of the time is spent file reading and parsing rather than sorting timestamp values. Since there are only 9321 possible pseudo timestamp values, which can be quickly sorted. While there may be millions of entry logs that need to be processed in many files. Therefore, this project will focus mainly on this part of the algorithm. Therefore, we will want to parallelize the file reading and parsing process as much as possible. In this section, we will compare the performance when using single threaded and multithreaded versions of file parsing.

Custom parsing of text data was used in almost all cases. Built-in parsing by high-level functions used only to measure performance and compare with hand-written.

Various functions and methods were used here to perform input, including: fgets, fread, multithread vs. single threaded, setvbuff with various options, and mmap.

Some definitions:

fgets: reads a line from the specified stream and stores it into the string pointed to by **str**. It stops when either **(n-1)** characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
fread: reads data from the given stream into the array pointed to, by ptr.
mmap: creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

fgets and **fread** are high level functions which perform reading from a file. They perform a buffering mechanism under the hood. But using it directly has drawbacks, since data will be copied twice. Once into a system allocated buffer, and a second time into a buffer allocated by us. We can avoid that by using **setvbuf**(file, NULL, _IONBF, 0); for fread. While that buffering mechanism for fgets it is quite beneficial, since the file will not be accessed in a line-by-line manner.

read is low level function which works directly with file descriptors. It allows to read pages using single system call.

Singlethread fgets vs fread

The simplest way of file parsing is to read all files in the directory sequentially. In each file, log entries are line by line, and then update the global counter array.

The functions used for this is `processfile(char *filename, int *counter)`, the first parameter takes the directory of the file, and the second parameter takes a pointer to the global counter array. A char buffer of size 40 to store each log entry, and then we parse the timestamp value using the hashing function h(x) = (x-min_timestamp) / 3600, where x is the timestamp being read. The result of the hash function is the index of the global counter array which is to be incremented.

Without setvbuff, (i) singlethreaded fgets takes 6.43 sec, (ii) singlethreaded fread without setvbuff, and buffer size 4029 runs in 9.21 sec.

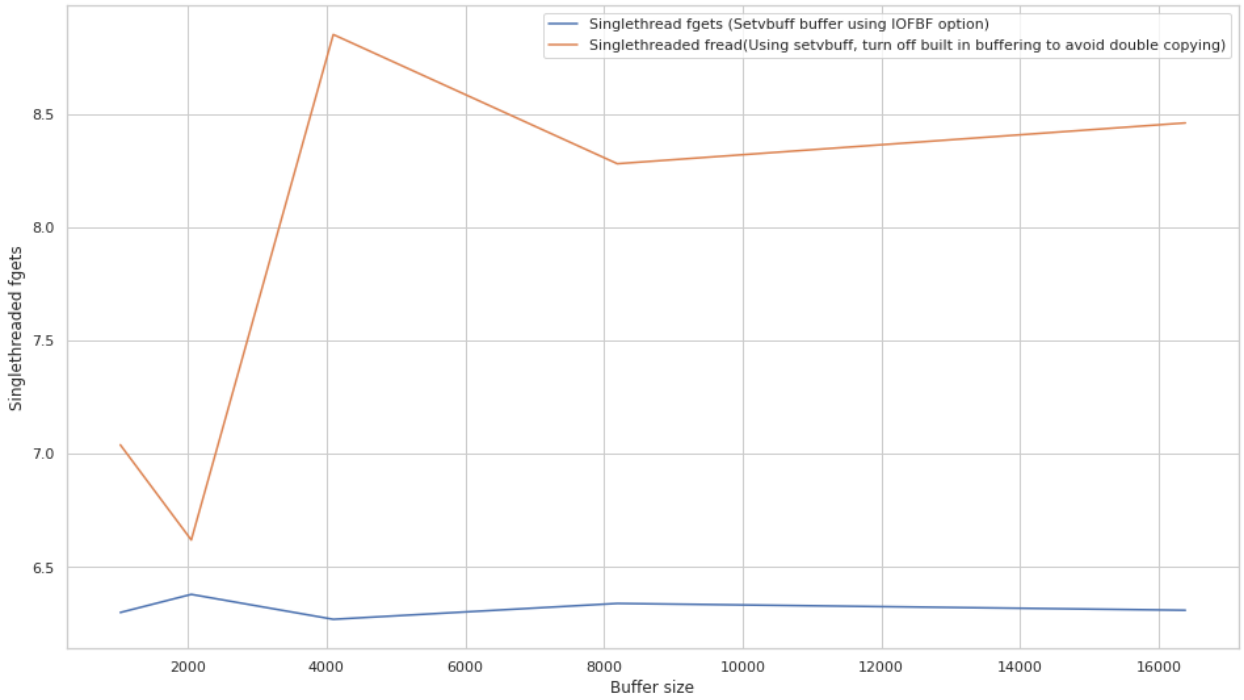

Figure 1. Execution times of programs with buffer size change.

	Singlethreaded fgets (Setvbuff buffer using IOFBF option)	Singlethreaded fread (Using setvbuff, turn off built in buffering to avoid double copying)
1024	6.30	7.04
2048	6.38	6.62
4096	6.27	8.85
8192	6.34	8.28
16384	6.31	8.46

Table 1. This table shows the runtime of the program using the 5 given test cases, with varying buffer sizes, and different setvbuff options.

From our observation testing singlethreaded fgets using varying buffer size, buffer size does not affect performance. Fread also seems to be slower than fgets and performance heavily depends on the size of that buffer.

Multithreaded fgets vs fread vs mmap vs read

Multithreaded version of fgets and fread parallizes file reading. The algorithm works by giving each thread a set of files to read, and they will be read and parsed in parallel. Each thread is assigned their own counter array, so they will update the number of occurrences of each timestamp in their local counter array. After all threads are finished updating their local counters, they are aggregated into the global counter. Our approach does not introduce race conditions, since threads do not need concurrent access to the global array, therefore no thread locking mechanism is needed.

The functions used for multithreading are `void startThreads(file_count, (char **)filenames)`, and `void *processfiles(void *arg)`.

startThreads takes two arguments, first is the number of files, second is the array of filenames in the directory. It has *localcounters[TNUM], which stores local counter arrays for each thread, and assigns a set of files for each thread to read and parse. The number of files each thread handles is defined by the following function:

block_size = (file_count+TNUM-1) / TNUM

New threads are invoked by calling processfiles(void * arg). The argument passed to each thread is a struct defined by:

```
struct thread_args
{
   int *counter;
   char **filenames;
   int start;
   int end;
};
```

The pointer to local counter for each thread is passed as *counter. Filenames is a pointer to the list of file names(paths to be precise) in the directory. Start and end are indexes to the filenames array, which indicates the files each thread will be handling. Besides that, this function basically does the same thing as processfile, reads and parses the allocated files to the thread sequentially. After all threads are finished, startThreads function aggregates all of the local counters into the global counter array.

In mmap approach, we designed algorithm that reads file in singlethreaded manner but parses it in multiple threads. Theoretically single file processing time should increase. This approach maps file directly into virtual memory space of the program and allows direct access of files. Since file is mapped directly, we can access contents multiple times, and therefore adjust chunks boundaries to match file content.

Number of threads

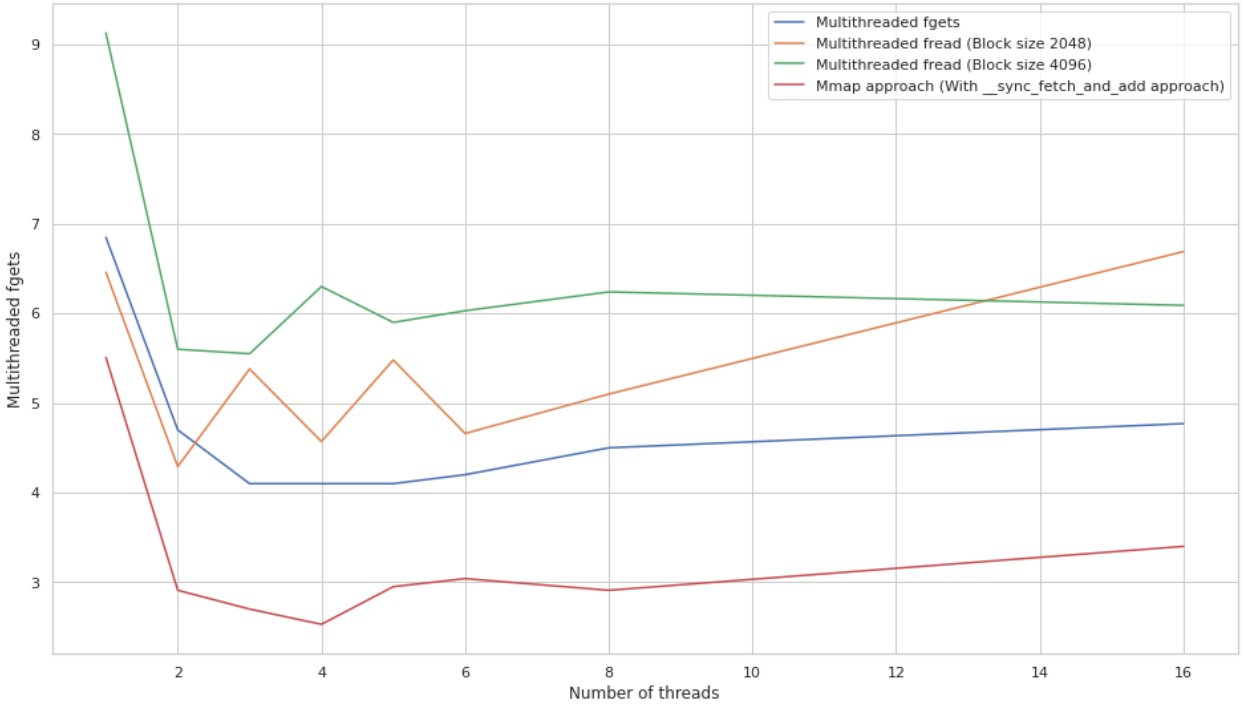

Figure 2. Execution times of programs varying by number of threads(less is better).

TNUM	Multithreaded fgets	Multithreaded fread (Block size 2048)	Multithreaded fread (Block size 4096)	Mmap approach (With ___sync__fetch__and__add)
1	6.85	6.46	9.13	5.51
2	4.76	4.29	5.60	2.91
3	4.19	5.38	5.55	2.70
4	4.13	4.57	6.30	2.53
5	4.19	5.48	5.90	2.95
6	4.26	4.66	6.03	3.04
8	4.53	5.10	6.24	2.91
16	4.77	6.69	6.09	3.40

Table 2. This table shows the results when we vary the number of threads, for each type of file reading and parsing approach.

From Table 2 and Figure 2, we can see that performance is the best when the number of threads is 4 for our approaches. Therefore, we conclude that the ideal number of threads is 4. This might be related to the hardware disk allowing effective parallel reading of only 4 files at the same time.

Mmap shows best performance, but it is exceeding memory limitations set by test script. Testing scripts measures virtual memory address space size, however actual memory usage of mmap solution smaller. But to submit solution successfully we decided to choose fgets based solution.

Thread locking mechanism

As previously mentioned, our approach to multithreaded file reading and parsing does not require locking. Nonetheless, we experimented with multiple locking approaches, which includes (i) local counter array of each thread, (ii) global locks after incrementing local counter array, and (iii) ___sync_add_and_fetch.

First approach is based on fact that our counter array is small - only 9400 elements. Therefore, we can allow to create copy for each thread and sum up results at the end.

First two approaches use 1 lock to protect access to the global counter array. (i) directly increments global array, (ii) increments local array but summing process is moved into the thread so if one thread finished reading and parsing faster than other it can sum up to global array. Third approach implements atomic instruction to access array element. The following table summarizes the performance when using the mentioned synchronization mechanisms.

Multithreaded fgets with TNUM=4,
Using local arrays, runtime is 4.13 sec
Using global locking during increment counter array, runtime is 19.39 sec
Using global locking after local array incremented, runtime is 4.25 sec
Usin ___sync_fetch_and_add, runtime is 4.89 sec

TopK Sorting Algorithm

Top k sorting algorithm was decided not to parallelize because it takes a very small fraction of actual runtime of the program. Also, parallelizing would bring additional problems, as the nature of topK is sequential, and if we solve topK for different arrays, the process of merging arrays to find global maxK is still relatively costly.

In our tests, even with K=9000, the topK algorithm performed in less than 2ms, usually around 0.001s, which is 3 orders lower than IO operations.

Due to reductions described in the Data Aggregation section, the problem is reduced to sorting of only 9000 items. Which can be done in almost constant time(O(N*log(K)) to be fair, where N is the number of different hourly timestamps (at most 9400), and K<=10000. As computers execute between 300-700 million instructions per second, we can see that it is almost constant).

For the topK, main functions used are TopK(), heapify() and compare_value_and_time() functions. We are implementing the min heap with the heapify(), because heap isn't a built-in function in the C.

Function heapify takes two arrays, one with the number of appearances (array counter) and one with the pseudo timestamps (array heap). Value n is the current size of the heap. We are using heapify() for creating the heap, by repeatedly calling it inside the buildHeap() function, and we are using it to iteratively go through each value in the heap[] array. By design, heapify() is achieving usage of pop(), then push(), and we are utilizing it for the update of heap from k+1-st element of the array until the last, nth element.

Function compare_value_and_time() is used as a comparison function. *values is the array of primary comparison (counter), and t1 and t2 are values for the second comparison function (heap[] in our case). If the numbers of appearances of 2 logs are different, the function will return a positive number iff the first log is a larger one. However, if they are equal, function will return positive number iff the first one is a later one (in terms of time appearance).

Function topK() is implementing the topK algorithm with the help of heapify(), buildHeap() and compare_value_and_time(). First part of the process is inserting the first k elements into the heap. This is achieved with the function buildHeap(). Then in every next step, we insert one element and remove the smallest element from the heap. This is achieved by heapify() function.

Thus, after t iterations, where t<=k<=n, we will have the highest k out of first t elements in the heap. When t=n we are done. For the last part, we could use heapify() with huge dummy insertion values to get topK elements from the heap, but we decided to use qsort() instead.

Compilers

Tested gcc and clang with option -O2, -O3
Size of the executable is not considered, only execution time is concerned.

	Single threaded fgets	Multithreaded fgets with TNUM=4
gcc	6.42	3.79
gcc -O2	6.10	3.72
gcc -O3	5.58	3.56
Clang	6.31	4.22
Clang -O2	5.72	3.64
Clang -O3	5.76	3.54

Table 3. This table shows average runtime of each file read and parse approach, with different compilers, with and without optimization flags (-O2, and -O3).

Code optimization

Code of the best solution can be further optimized by utilizing various optimization tricks and techniques. Code wasn't written in this way initially because it has less portability and readability.

Important to note that code optimization is only useful in tight loops because the program is mostly run in those parts of the code.

Timestamp parsing. One of the most significant improvements was achieved by changing built-in atol and strtol functions with custom macros utilizing loop unrolling. We simply read 10 characters and convert them using simple arithmetic.

More improvement in this direction can be achieved by skipping last 2 digits of timestamp we are reading. We later divide it by 3600, so 2 digits at the end are not significant. During the tests with 50 runs of all 5 test cases it has shown 3% decrease in execution time.

```
long time_stamp = strtol(buffer,&temp,10);
```

Becomes

```
at = buffer;
PARSE_FIRST_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
PARSE_NEXT_DIGIT
time_stamp *= 100;
```

```
#define PARSE_FIRST_DIGIT  \
        time_stamp = *at++ - '0';
#define PARSE_NEXT_DIGIT              \
        time_stamp = time_stamp * 10 + *at++ - '0';
```

Use of macro. In general, calling of function requires manipulation with stack and reallocation of some variables. Introduction of macros in most intensive parts of program helped to reduce execution time.
For example, compare_value_and_time() function to compare values when rewritten as macros becomes:

```
#define COMPARE(v, a, b) ((v[a] != v[b]) ? (v[a] < v[b]) : (a < b))
```

Increasing locality. In general, code might be swapped from memory while executing and loading it back takes some time. By placing the most related and most intense part of the program together helps to reduce execution time. This is hard to measure since code was initially written in a convenient way.

Custom ceil. Instead of using ceil function call in <math.h>, we used regular arithmetic. i.e. (a+b-1)/b instead of ceil((double)a/b).

Remove branching when unnecessary. Instead of using if statement in loops we can create 2 separate versions of file to minimize branching effect especially in reading and parsing parts of code.

Using unsigned before division by int constant. This helped to reduce number of instructions since now division doesn't have to take into account sign of the long value type.

```
counter[(time_stamp-start_timestamp)/3600]++;
```

```
228     sub     rax, qword ptr [rip + start_timestamp]
229     add     rax, r12
230     imul    r13
231     mov     rax, rdx
232     shr     rax, 63
233     sar     rdx, 10
234     add     rdx, rax
235     add     dword ptr [r14 + 4*rdx], 1
```

Figure 3. Instruction of counter[(time_stamp-start_timestamp)/3600]++; (Clang -O3)

```
counter[(unsigned)(time_stamp-start_timestamp)/3600]++;
```

```
226     add     eax, -1793725248
227     imul    rax, r12
228     shr     rax, 43
229     add     dword ptr [r14 + 4*rax], 1
```

Figure 4. Instruction of counter[(unsigned)(time_stamp-start_timestamp)/3600]++; (Clang -O3)

We also thought about converting

```
counter[(unsigned)(time_stamp-start_timestamp)/3600]++;
```

to

```
counter[(unsigned)(time_stamp-start_timestamp)/36]++;
```

By skipping reading the last 2 digits of the timestamp. But this didn't affect the performance of the program. So we chose to use more readable version.

All those optimizations helped to reduce runtime of a program furthermore, and with Clang -O3 compiled option we were able to reduce execution time from **3.53 to 2.31** seconds.

Compiling code

Run the following to compile ptopk.c:
```
clang ptopk.c -o test.o -l pthread
```

Compile with optimization flag:
```
clang ptopk.c -o test.o -l pthread -O3
```

It is important to compile programs using optimization flag because our code was optimized for this compile command and compiler to reduce number of cycles CPU run in most tight loops.

Conclusion

Overall, it was hard to measure code's performance since it depends on the status and load of the server. Therefore, we conducted all measurements in the 5am period by assumption of minimal interference from other users.

For compatibility of our solution, we replaced qsort at the end of topK procedure with custom heap sort algorithm due to strange behavior of code when submitted. This lead to increase of execution time by 100ms as tested on local test cases.

We were able to simplify program and memory constraints by using math and reducing problem size. This gave us more freedom in terms of topK algorithm implementation since bottleneck of the program was reading and parsing and sorting algorithm itself took only 1-2ms.

We decided to use process single file using single thread, because sequential access of the file is faster and easier to code due to no need of synchronization of single file read. But multiple files are processed in different threads. This allows us to utilize parallel reading capabilities to reduce overall processing time.

As was identified in our evaluation, mmap approach gave us best performance in terms of execution speed, but virtual memory address space increased over 600MB, which is not acceptable by testing script. Therefore, we used multithreaded fgets approach which also have shown good performance. Number of threads was chosen during the experiments by comparing how program behaved during read. Underlying implementation of fgets high level function uses buffered read, but setvbuf function have shown no visible effect on performance of the program.

After making architectural choices of the code, we optimized code performance further. Reducing processing time in some of the loops was very important since some of the lines executes millions of times.

As the final solution we submit the file, which finished all available test cases in 2.31 seconds. By our measurements.