CMPE 382 HW3 Report

Berke Öcal

Berkay Özek

TASK 1

We have implemented a ternary search trie for all of the tasks. We have taken the implementation from https://www.geeksforgeeks.org/ternary-search-tree/. Additionally, we have modified the ternary search trie to keep track of terms frequencies and added several other functions such as trie deletion and a different insertion function which we used at task 5.

	Execution Time
Task 2	23,002 s
Task 3	22,432 s
Task 4	30,736 s
Task 5	10,004 s
Task 6	20,277 s

	Ram Usage
Task 2	1.1 GB
Task 3	1.1 GB
Task 4	1.1 GB
Task 5	2.6 GB
Task 6	1.1 GB

TASK 2

Implementation of this task is pretty straight forward. We have declared 10 file names and 10 file pointers. We opened 10 of the input files to read the input files. We have read input lines with fgets() function line by line. Then we used strtok() to parse the input string and inserted the queries to ternary search trie. Lastly, contents of ternary search trie are printed out to “dictionary2.txt” file. There is not any comparison to be made as this is actually the first task we have implemented as a dictionary.

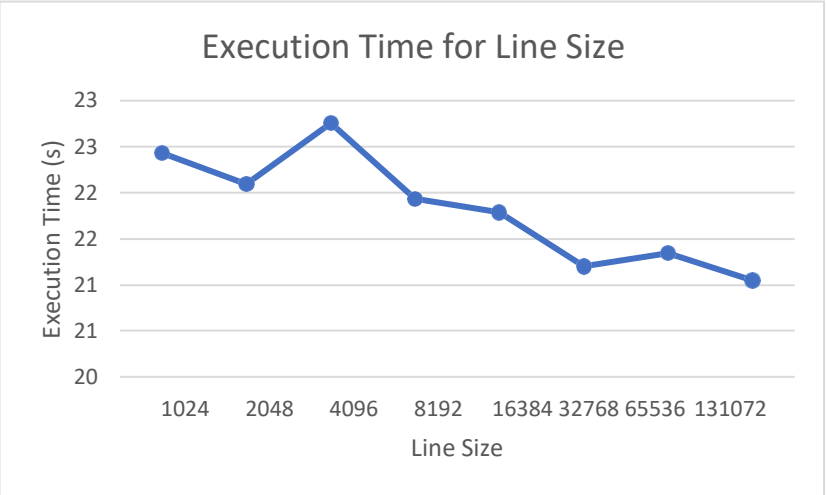
Execution time of the task 2 is 23,002 seconds. Note that measured time depends on several factors such as CPU load and hardware. Hence, it is nearly impossible to get the same execution time with different program executions.

TASK 3

Third task is similar to the second task. Only difference is that we are reading input as chunks of bytes instead of reading line by line. We had to observe whether reading a single item from disk or multiple items benefit our code or not. Our implementation had different

LINE_SIZE values for reading LINE_SIZE number of bytes. The graph and table below show the execution time and LINE_SIZE relations:

Line Size	**Execution Time**
1024	22,432
2048	22,095
4096	22,754
8192	21,936
16384	21,788
32768	21,204
65536	21,347
131072	21,052



It can be observed that our program takes less time to execute as we increase LINE_SIZE. Reason for this is we perform less I/O operations and less disk accesses. Furthermore, CPU utilization increases, and our program runs faster. A simple mathematical comparison would be as following: Assume that an average single line has 120 characters which makes 120 bytes total. In order to read 1200 bytes, we will have to call fgets() function 10 times. However, we can read 1200 bytes with a single fread() call and partition the input string in the main function. Data collected in this task supports that this is the case. Reading input with larger chunks of bytes increases the program speed as there is less overhead and less resources are spent on I/O operations.

Note: Sometimes task 3 performs slower than task 2. This is heavily dependent on LINE_SIZE variable in the code.

TASK 4

Input operations at the task 4 are implemented with fgets() function. We actually used our code from the task 2 and we have added threads and a single mutex. Mutex is essential because all of queries were stored in a single trie, and this situation causes that single trie to be a shared resource which is a critical section in our code. Before the usage of the mutex, threads were accessing the same trie asynchronously. Hence, terms frequencies were inaccurate and “dictionary4.out” file was a complete mess. Mutex resolved this issue, and we got the same “dictionary4.out” file as the task 2.

Execution time of the task 4 with 10 threads and a single trie is 30,736 seconds. We are using threads, but our execution time is much slower than previous tasks. Reason for this is only a single thread can insert to the trie in any given time. Hence, other 9 threads must wait for that insertion completion and mutex to be unlocked. This also reduces the CPU utilization because I/O wait operations take much more time. This task is costlier than previous tasks because there is more overhead. There is the cost of context switching for threads and it adds up to the execution time. These problems are main reasons why this implementation and task is not very efficient.

TASK 5

There was no need to implement a mutex as there was not any critical section. Each thread had their own respective ternary search trie. We noticed that construction of the tries takes most of the execution time. Yet, when threads did not have to wait for each other and were able to operate without mutexes we got the fastest execution time.

Execution time of the task 5 with 10 threads and 10 TST's is 10,004 seconds. This result is the best we have got so far. Reason for this is the main operation that takes most time is insertion to the trie. Construction of a single trie for all of the input files is the costliest part of the program. Implementation of the threads with multiple tries reduced the workload of a single trie. Depth of 10 tries is a lot less than depth of a single trie implementation. Hence, every insertion operation is faster. On top of this threads are working independently from each other as there is no critical section. Furthermore, CPU utilization is the highest because multiple cores are running (We used 4 CPU cores). This task requires more computational power and memory, but the execution speed is the best result we got. Constructing tries individually with their respective threads made a drastic difference in execution speed.

TASK 6

Execution time of the task 6 is the fastest among all tasks with 20,277 seconds. Of course, execution speed is not as good as task 5 but we can still learn from this situation. Execution time of task 6 is lower than any other task but task 5. Reason for this is, there is a single disk access for every file. There are not any additional disk accesses. This means that CPU utilization is high and wasted time on I/O accesses is less compared to task 2, task3 and task 4.

TASK 7

Our initial implementation of the R-way trie did not function properly. R-way trie consumed too much memory and we were not able to read all of the queries. We were only able to get the letters from the alphabet. This caused several problems. First of all, words that contained characters from the extended ASCII table were parsed incomplete as we could not process extended ASCII characters. Secondly, we should have shortened the input files, but we actually did not want to do so. We wanted to implement all of the queries to get a better view of the project and the tasks. In order to overcome these difficulties, we have implemented a ternary search trie. Initially, 12GB ram was not enough for 10 input files. After the ternary search trie implementation each task consumes less than 3GB of ram. This was a major improvement for us as we were able to process the queries and implemented the threads as we wished. Solving the memory issue also helped us with threads. Task 5 consumes nearly double of the memory of the other tasks. We would not be able implement task 5 without this memory optimization. Even if we managed to implement it somehow it would not be suitable for comparison with other tasks as we would most likely have to change the input files.

General Summary

We observed several different things during this project. Disk accesses are costly, and they hinder execution time. Accessing RAM or cache is considerably faster than accessing the disk. Multithreading with multiple cores increases the performance drastically because of parallelism. We learned how multithreading and memory accesses effects program performance.