CMPE 382 PROJECT 3

Operating Systems Project #3:

Creating Dictionary for the AOL Dataset

Group Member

Ahmet Kaan TOPRAKÇIOĞLU - 12742035240

Tanercan ALTUNTAŞ - 12595552970

INSTRUCTOR: Dr.Tayfun KÜÇÜKYILMAZ

Index

Introduction

During this project we learned about multiple topics, like modifying an existing code base, using threads along with disks, understanding and observing how computer disks work and experiencing high performance computing and it's intricacies in a real life problem

The task was to build a dictionary of AOL web queries, with the number of times that each query is seen. We used thousands of queries separated in ten files, each file over 60MB in size. The idea of using such an input set is to deal with problems that only arise during real world scenarios.

As we mentioned, this project made us change our original code several times, adding new features and changing the way we attacked the problem, and comparing each version with the previous looking for performance differences.

We also had the task of implementing a data structure called "Trie", commonly used for tasks like the one on this project. A "Trie" or "prefix tree" allows us to store strings with fast insertions and also lookups, but most implementations only keep track whether a string is present or not, and ignore repeated insertions.

Project Structure

We made a little build script - “build.sh” - for building all programs, which will be placed inside the “bin” folder. Then, each program can be executed passing as argument the path to the dataset folder.

Each program is named after a task, and we also include a single “trie.h” file shared by all programs, which contains our Trie implementation with some useful operations.

All programs were compiled with GCC 10.3.1 with maximum optimization (-O3)

Our Test Machine

All programs were executed on the same machine with the following specifications:

- Intel I5 6500 CPU

- Kingston 8GB RAM DDR3

- Crucial BX500 240GB SSD

- Ubuntu 20.04 OS

Task 1: Building a Memory-Based Data Structure for Building the Dictionary

Regarding the Trie, it’s common that this structure only tells if a string is present, and repeated insertions are ignored. In our implementation, the Trie keeps track of the number of times a string has been inserted - the frequency. Our initial implementation also used an array of pointers to store the next character, following the traditional implementation. We found that this required a lot of memory and we changed our implementation for a Linked List based one, reducing the total memory consumption at the cost of slower insertions, allowing our program to run on our test machine.

Task 2: Sequential Execution

Once we implemented the Trie, implementing the first task consisted of translating the pseudocode in the project description to C. Soon we got our implementation running and we got an average execution time of 37.5s (+/- 1s).

Task 3: Sequential Execution - Multiple Queries

For this task, instead of reading single lines from each file we read blocks of growing fixed and we compared the result. The idea is that by reading larger blocks we reduce the time spent on disk I/O, reducing the overall execution time. We note that we didn't consider truncated lines at the end of a block.

With different block sizes (in bytes) we got:

Size (byte)	Operation Time (s)
1024	38.2s
2048	37.7s
4096	37.5s
8192	37.2s

with some margin of error - (+/- 0.5s).

We did get better results, but not by much. It's possible that the cause of this is that we're using an SSD with fast read times.

Task 4: Threaded Execution

Then, we used threads in order to reduce the execution time of our program. Given that we are dealing with multiple files, naturally we separated the processing of each file in separate threads, gaining parallelism. Note that even though each thread had its own file, all threads used the same Trie, which resulted in incorrect results: for example, some queries didn't appear, or the frequency of some query was incorrect. Then, we used a mutex to prevent concurrent mutations of the Trie, but we ended up with really poor performance: our program took over 1m30s.

Task 5: Threaded Execution - Multiple Tries

This version involved the use of individual Tries per thread, eliminating the need for locks since there is no shared state between threads, but adding the need of merging the Tries into a single final Trie. This version was the fastest, with an execution time of 25.2s.

Task 6: Completely Memory-Based Dictionary Creation

The last task involved reading each file fully into memory. We expected similar results to our initial sequential version since we knew that disk I/O was not a big factor in the execution time. This version ran in 37.0s

Task 7: Improvement

A possible improvement is to move the processing of each file to a separated processor like a GPU. We found that there are GPU based implementations of Tries, so it's possible that by using such devices we can get extremely fast performance.

Such Trie implementation can be found on this paper:
https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5027