Tal Derei

CSE 375: Assignment 1
Bank Application

Design Framework:

When designing the framework for my application, there were a number of design approaches in mind: **[1]** Map of Vectors -- e.g. static std::unordered_map<std::vector<K>, std::vector<V>> map, **[2]** hashmap that maps to separate vectors (i.e. you do all the account deposit and balances on the vectors, and then map the results to a hashmap), **[3]** Map - perform the computations, locking, threading, on a single data structure without the use of vectors.

When designing these solutions, you could do everything as a single data structure, or design a data structure with an application layer that uses that data structure. In the sequential code, design approach **[3]** was implemented as a baseline to do all insert/remove/lookup/deposit/balance API calls on a single hashmap. In the concurrent code, a separate design was used: **[4]** Each 'bucket' struct contains (1) a mutex lock, and (2) an empty unordered_map. Then you implement a vector (or unordered_map) of 'bucket' objects. Each bucket corresponds to a subset of the total number of accounts, and every key/value pair in a bucket corresponds to a separate account. The system design incorporates locking entire buckets, performing some operations on the accounts inside that bucket, and then releasing the lock on the bucket. The vector of buckets acts as a containerized wrapper for the unordered_map.

There are two more approaches I'd like to quickly highlight that were in consideration as well: **[5]** Rather than locking the bucket's themselves, somehow lock the individual key/value pairs (i.e. the accounts) themselves. **[6]** You can replace the 'vector' data structure in implementation **[4]** with an unordered_map, which would ideally provide speedups. It would essentially be an unordered_map of structs of unordered_maps.

Portions of the code design were implemented using the CSE 303 framework from P2

Performance Plots:

[see next page]

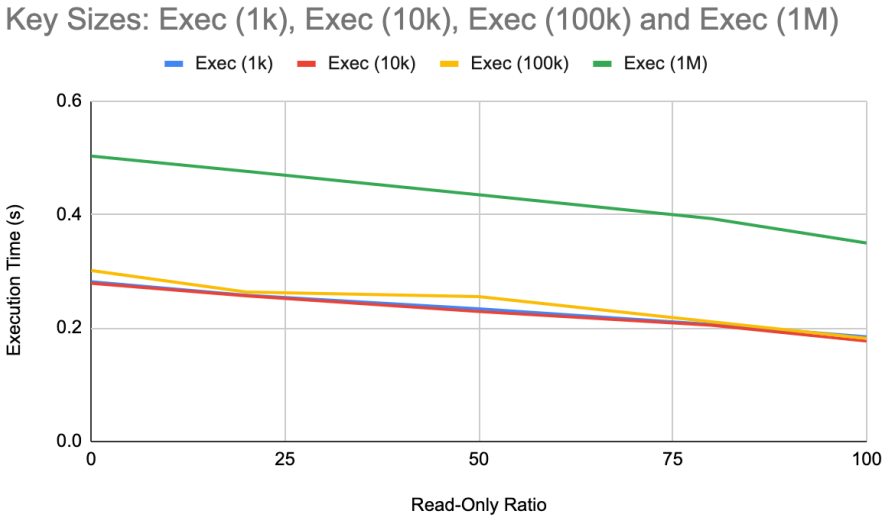

Figure 1: Sequential Execution of Insert/Remove/Update/Lookup APIs

In the plot above, I decided to implement a test checking the execution times of the insertions, removals, updates, and lookups associated with the unordered map in the sequential execution. This would provide a good baseline in determining whether the map was implemented correctly, and help in identifying whether potential bottlenecks in the sequential execution were associated with these API calls, or the deposit() and balance() functions. The bottlenecks were found in the latter. Different key sizes (1k, 10k, 100k, 1M) were implemented as a function of read-only-ratio (i.e. what percentage of API calls were lookups) and their execution time. Across the board, as the read-ration increased, the execution times decreased proportionally for all key sizes. Additionally, the unordered maps were pre-initialized to 1/2 of the max key range, and each test ran on an 1M iterations. The functionality was assumed to be correct as the unordered_map maintained its original size after these random insert/remove/lookup operations were complete.

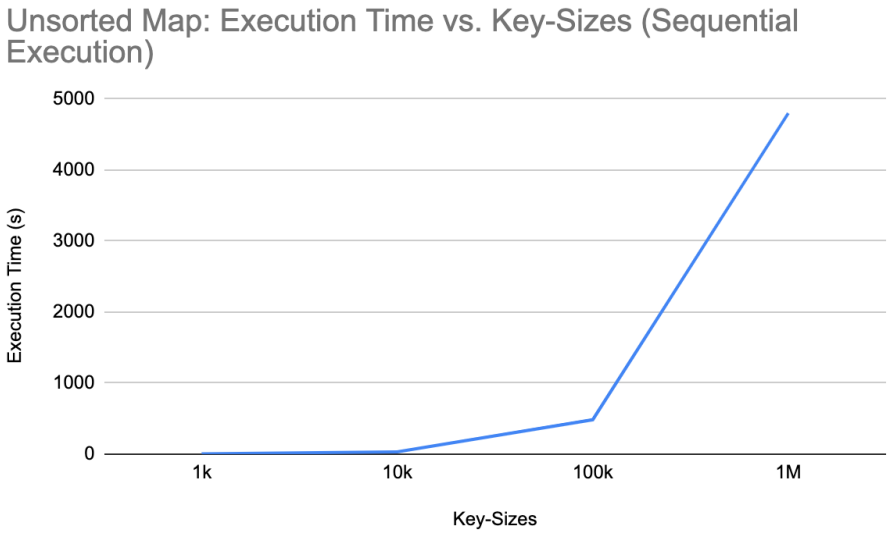

Figure 2: Sequential Execution of Unsorted Map (Balance and Deposit APIs)

The plot above highlights the sequential execution time (one thread) for balance and deposit API calls, on a range of 1K to 1M keys and 1 M iterations. The execution times exponentially increased as the key sizes increased beyond 10K. One hypothesis for these slow execution times was that the deposit() call's time complexity is inefficient. The deposit function randomly chooses two buckets in the vector of buckets, and then randomly selects two accounts (one from each bucket) using std::advance. Std::map iterators are bidirectional, which means selecting a random key (i.e. account) will be O(n). As the number of keys scales, this performance gets worse, because you're bound by the limitations of the vector.

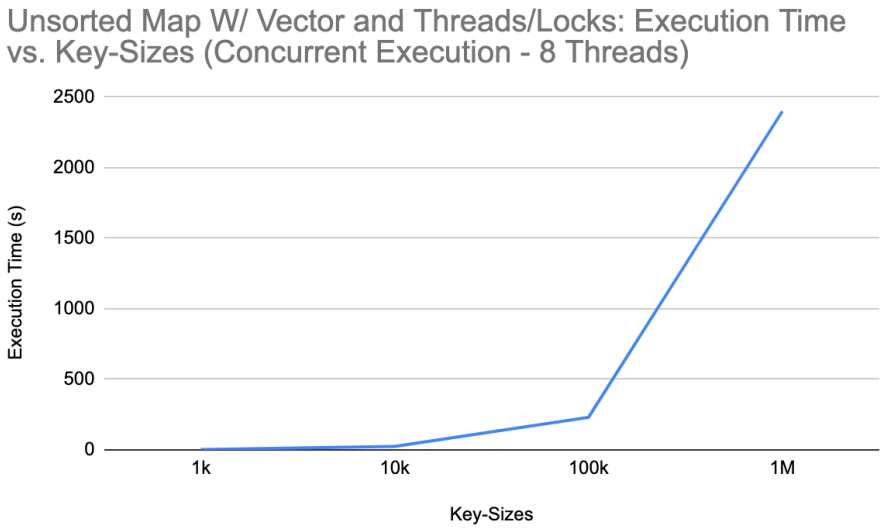

Figure 3: Concurrent Execution with Locking/Threading (Balance and Deposit APIs with 8 Threads)

The plot above highlights a concurrent execution of the codebase with mutexes and multithreading (8 threads), but the performance results aren't much better. The worst-case execution times as the max key size increases has been cut by only half, rather than a theoretical full 8x parallel speed. This implies that (1) there are some sequential portions of the codebase that can't be parallelized, or (2) the interaction between parallel threads and locks wasn't implemented correctly. The latter seems more plausible. The overhead of spinning up the threads, and thread contention amongst the different locks seems to be causing this bottleneck in performance on the multithreaded application.

Next Steps:

[1] Try-Locks: Need to incorporate a try-lock system. Right now, I lock all the buckets, perform the balance/deposit operation, and then unlock all the buckets. Try-lock would allow me to implement a system that only locks two buckets where the operation is taking place.

[2] Lock Table: Need to incorporate a lock table to handle locking logic.

[3] Queue: Implement a queue (global space) to dynamically load balance the locks.