I began by reading the chapter in the textbook which was very useful to understand how the algorithm works. While implementing my sequential program, I made some assumptions regarding resizing. I simply doubled (or halved) the size of the table. I resized the table after a cycle was detected. I followed a naive approach by simply resizing after 10 swap iterations. I thought about keeping track of the seen indices. However, I figured that the added memory and time to determine that would be much more than the small cost of 10 iterations. In addition, I shrunk the table when the number of elements was less than 25% of the capacity. Some testing could be done here to determine a more optimal decision. To test the program, I varied the range of elements in the table from 100 to ~ INT_MAX / 2. This allowed me to compare how the application scaled as the number of elements increased. The table was filled with half the range and then tested with the 80/10/10 split. As you can see, the application scaled fairly linearly as elements increased (Figure 5).

For the concurrent program, I began by implementing the algorithm described in the textbook. The textbook implementation used a per-bucket lock, so it was essentially fine-grained. This implementation was not slow but was significantly slower than the sequential version. Then, recognizing the read-heavy nature of testing simulations, I implemented a r/w lock to hopefully exploit the advantages of a shared_mutex. However, I recognize that since I only utilized a single coarse-grain lock, this was effectively only slightly better than using a single coarse-grain lock. The next implementation used lock striping to hopefully reduce the lock contention. Unfortunately, I did not see the expected results. For my final implementation, I recognized the speed of the sequential version. So, I came up with a solution of a table of sequential cuckoos. Essentially, I created a wrapper around the sequential implementation where each value is mapped to an index in a vector of sequential cuckoos, where it is added. I implemented concurrency by controlling access to each sequential cuckoo. With some parameter tuning, this version saw speedup in comparison to the sequential version at 24 threads. The sequential version took ~8.7 seconds to run while the concurrent version took ~8.2 seconds. Something that was interesting is that running 100% contains was faster on the sequential version, regardless of implementation. This indicates to me that the real bottleneck is the read operations, as this is the majority of the operations.

For the transactional version, I first read up on the documentation from cppreference to understand how the feature worked. Then I identified the smallest possible critical section to reduce the atomic region. Finally, I wrapped the necessary regions with synchronized/atomic. I also attempted to implement a lock-free contains/remove using logical deletions. My goal was to reduce overhead for those methods by using an atomic boolean and compare and swap. However, I was not successful and do not believe my program is technically correct.

As you will notice in the results, for large enough N, I did achieve speedup with multi-seq. In addition, multi-seq and transactional scaled fairly well as threads increased. For multi-seq, the number of sequential buckets used heavily influences the runtime. Proper testing should be done to determine optimality based on the application.

Figure 1:

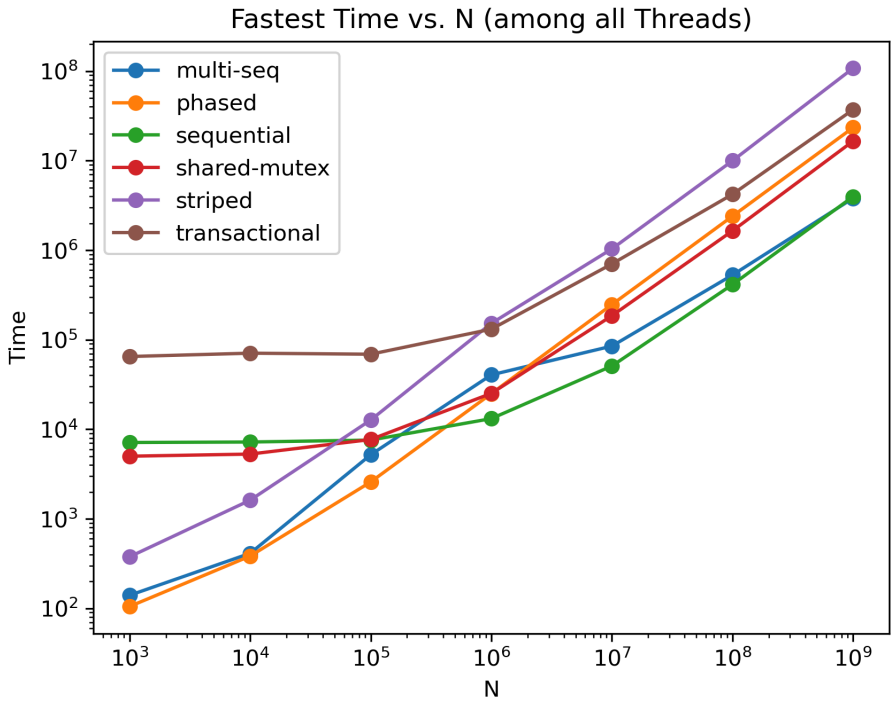


Figure 2:

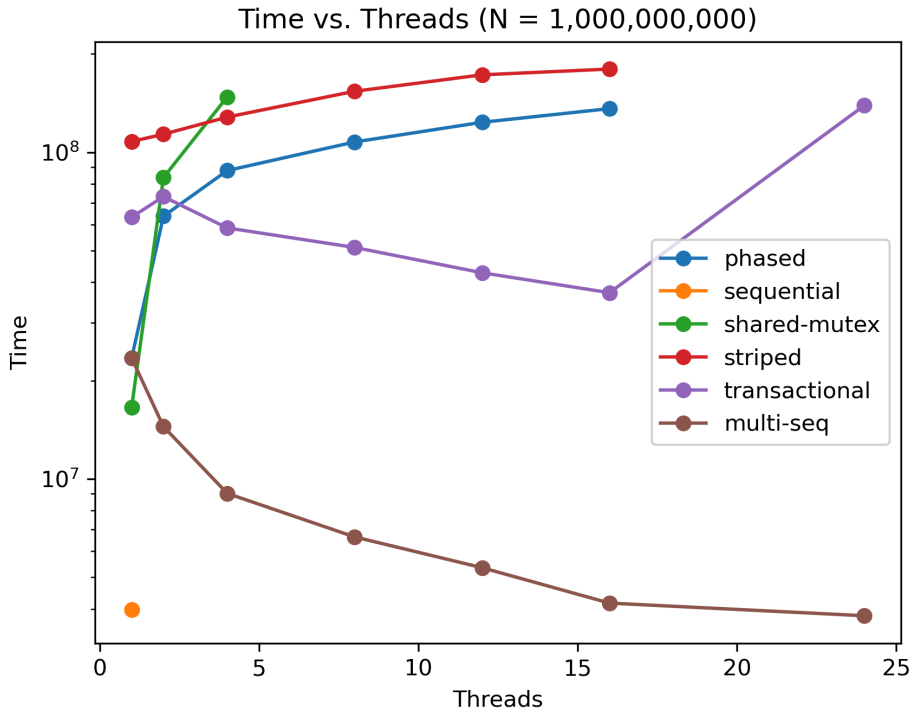


Figure 3:

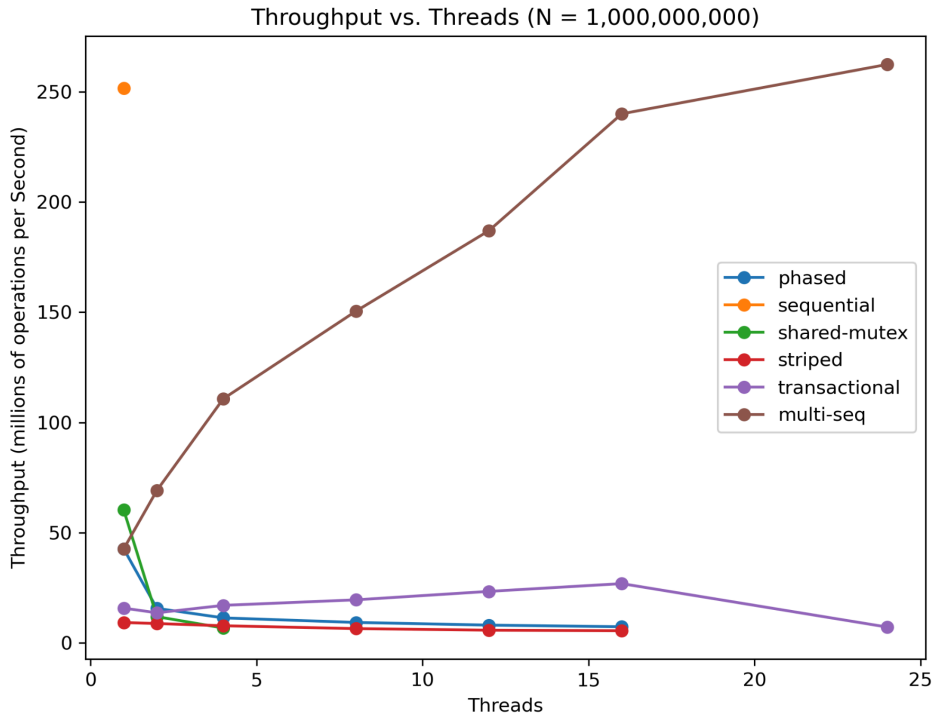


Figure 4:

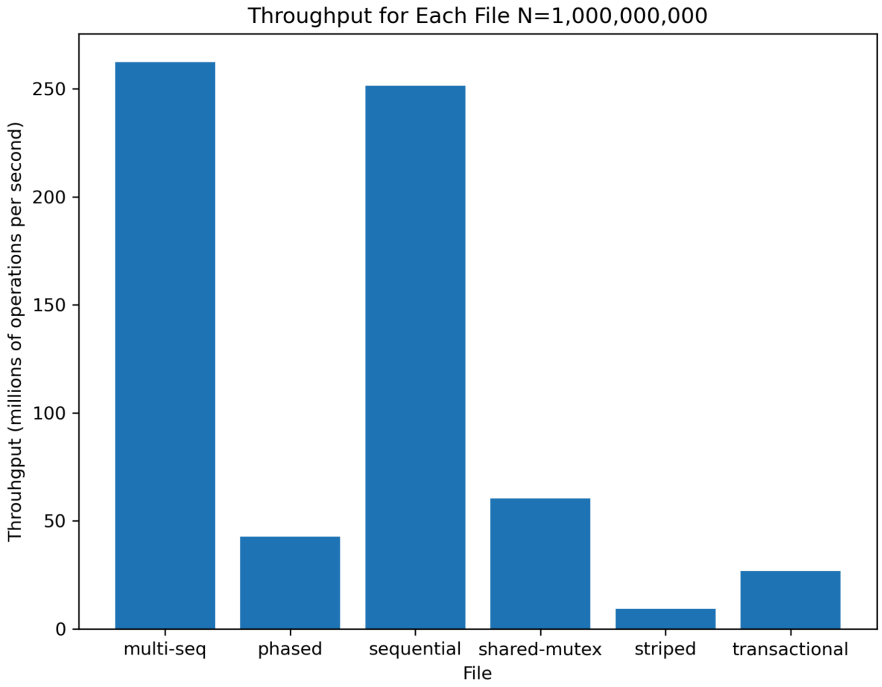
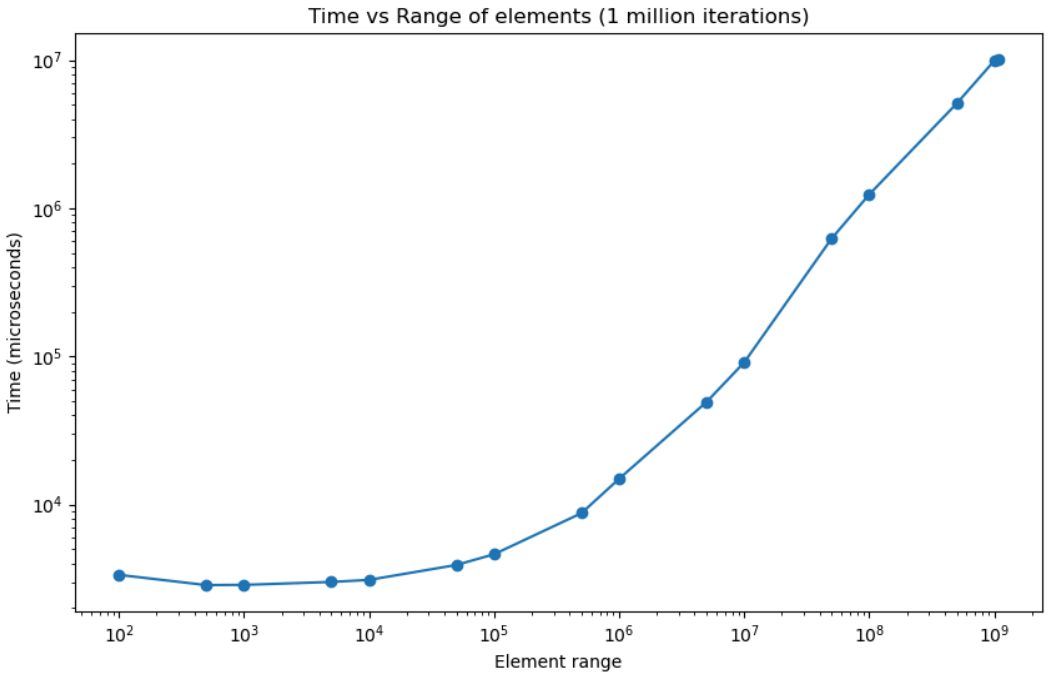


Figure 5:


Time vs Range of elements (1 million iterations)
Time (microseconds)
Element range
10^7
10^6
10^5
10^4
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9