For ease of testing, I structured the main() in the same way for each implementation. Doing so allowed me to create one generic shell script to tabulate results. Each main() parses the command line inputs (# of threads, accounts, and iterations), calls insert(), creates the threads that call do_work(), and prints the results. I also made use of #ifdef/ifndef for easy testing.

I began by implementing the sequential program because I believed it would be the easiest to implement and would serve as an easy framework to model the rest of my solutions. After the sequential, I attempted implementations in the following order: coarse grain, fine grain, reader/writer locks, buckets, and buckets with reader/writer lock. I made use of the reader/writer lock because the time spent reading (balance()) is far greater than deposit() which takes almost 0 time.

As seen in Figures 1-3, the sequential implementation consistently performed better in all cases. As I increased the number of threads duration grew linearly except for the bucket implementation. In this case, duration closely followed fine-grain locking. This occurred because as threads increased, competition for the few locks grew. What I found interesting, was that as I increased the number of accounts, duration increased. This doesn't make sense, especially for fine-grain locking, as there would be less competition for the locks. One possible reason is that increasing the number of accounts increases the time for balance() which takes the vast majority of the time. In an attempt to fix this, I implemented an atomic balance() counter representing the number of balance threads waiting. If a deposit() sees that the counter is greater than 5 it would yield until the number waiting was fewer than 5. This resulted in the marginal improvements. Finally, in Figure 2, you can see how iterations affected the duration. Duration amongst the implementations is approximately the same for < 10^4 iterations. However, for 100,000 and 1,000,000 iterations, we find a clear ordering where sequential comes out on top and fine grain on the bottom.

I was also interested to see how blending reader/writer locks and buckets would perform. I discovered that this implementation was the quickest. From this, I wanted to find the optimal number of locks (or buckets). Figure 4 depicts this with 1,000,000 iterations and 1,000 accounts. As locks increased, duration decreased until ~70 locks at which point duration stayed slightly below 4750 ms.

Figure 1:

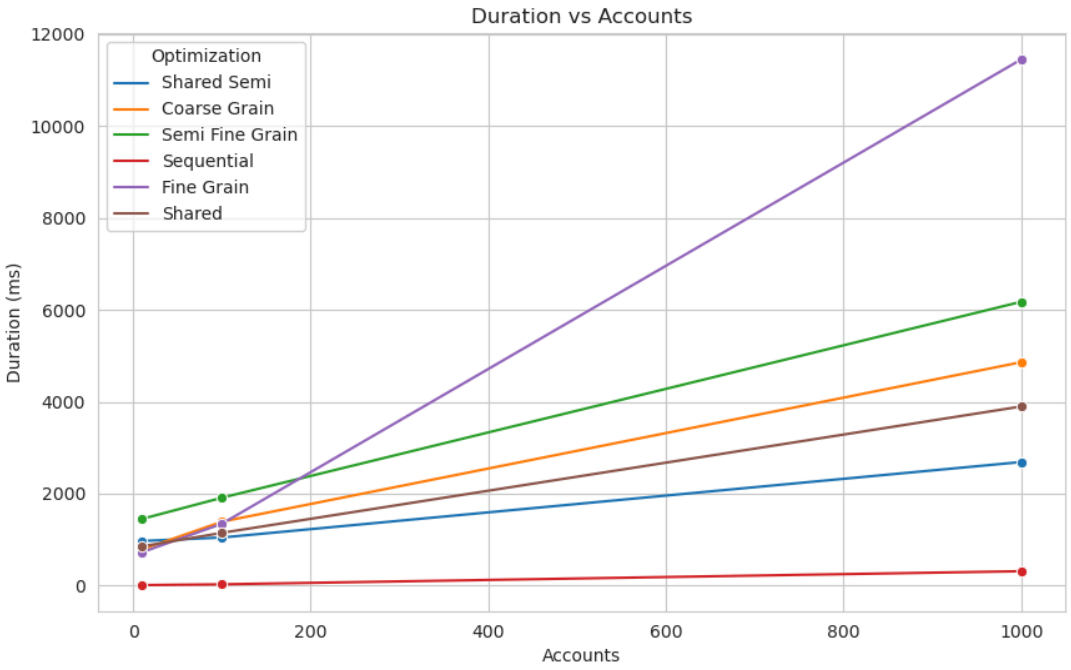


Figure 2:

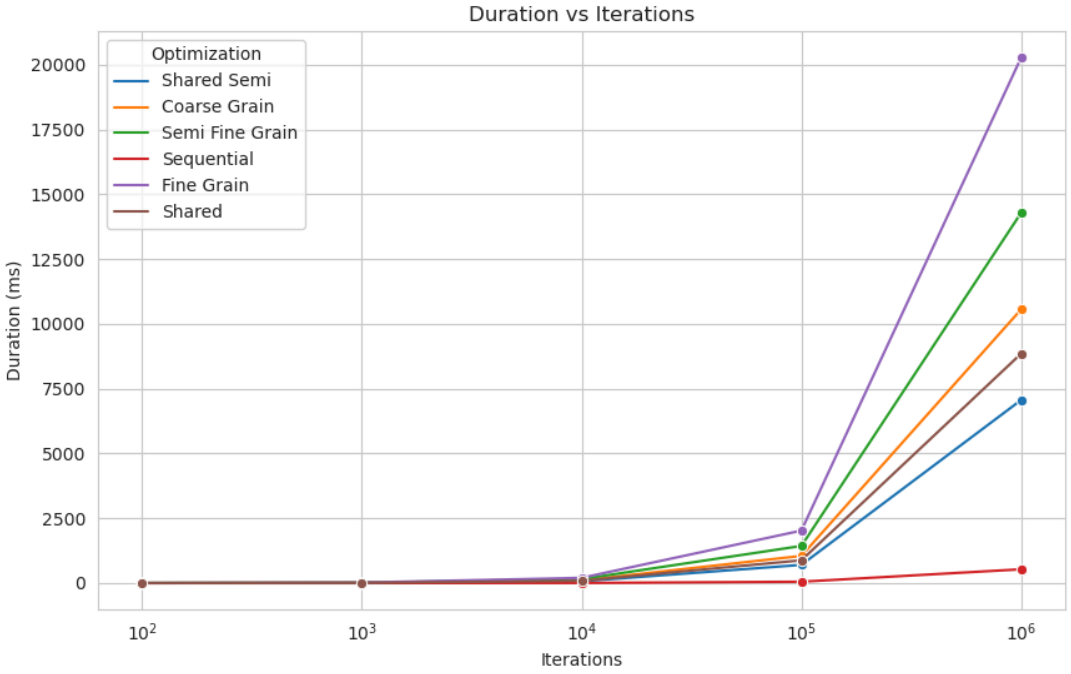


Figure 3:

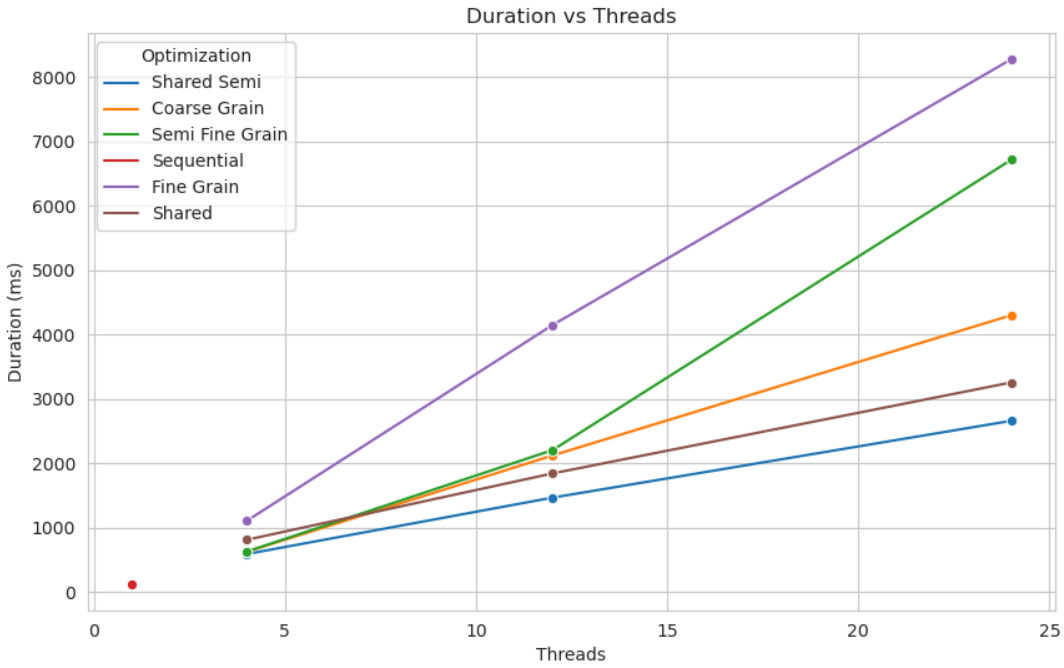
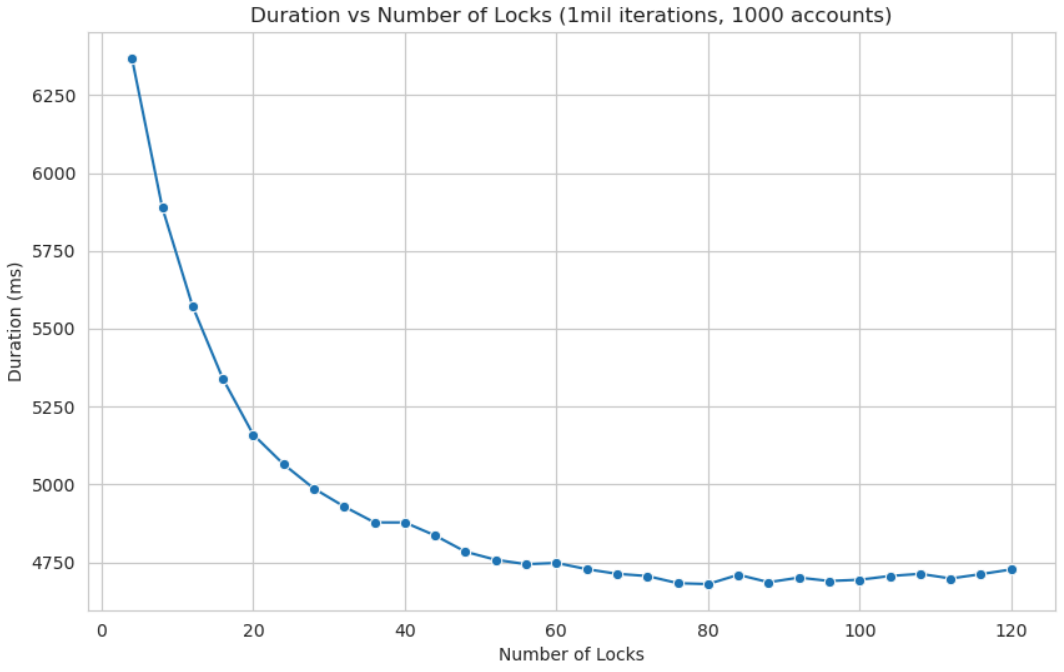

Duration vs Threads
Optimization
Shared Semi
Coarse Grain
Semi Fine Grain
Sequential
Fine Grain
Shared
Duration (ms)
Threads
0
1000
2000
3000
4000
5000
6000
7000
8000
0
5
10
15
20
25

Figure 4:


Duration vs Number of Locks (1mil iterations, 1000 accounts)
Duration (ms)
Number of Locks
4750
5000
5250
5500
5750
6000
6250
0
20
40
60
80
100
120