Time Spent: 60 hours

I will be referring to goroutine as gr and goroutines as grs in this lab report.

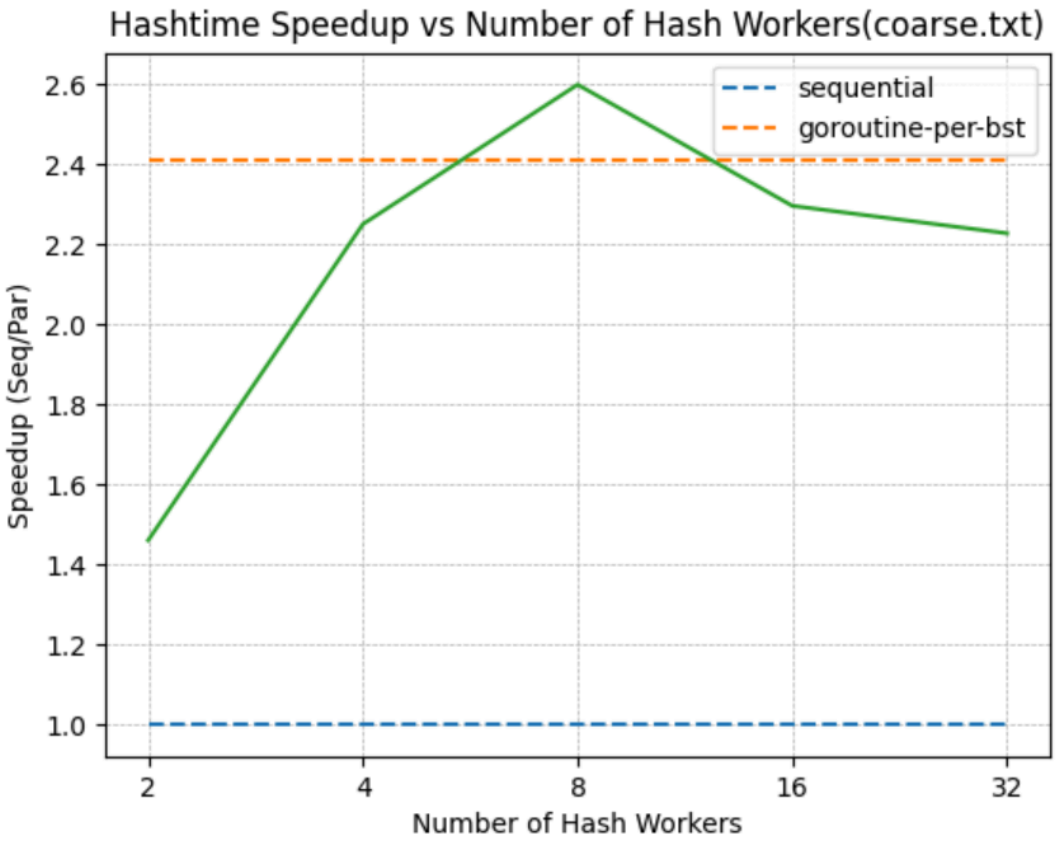

Figure 1a. Hashtime Speedup Graph(coarse.txt)

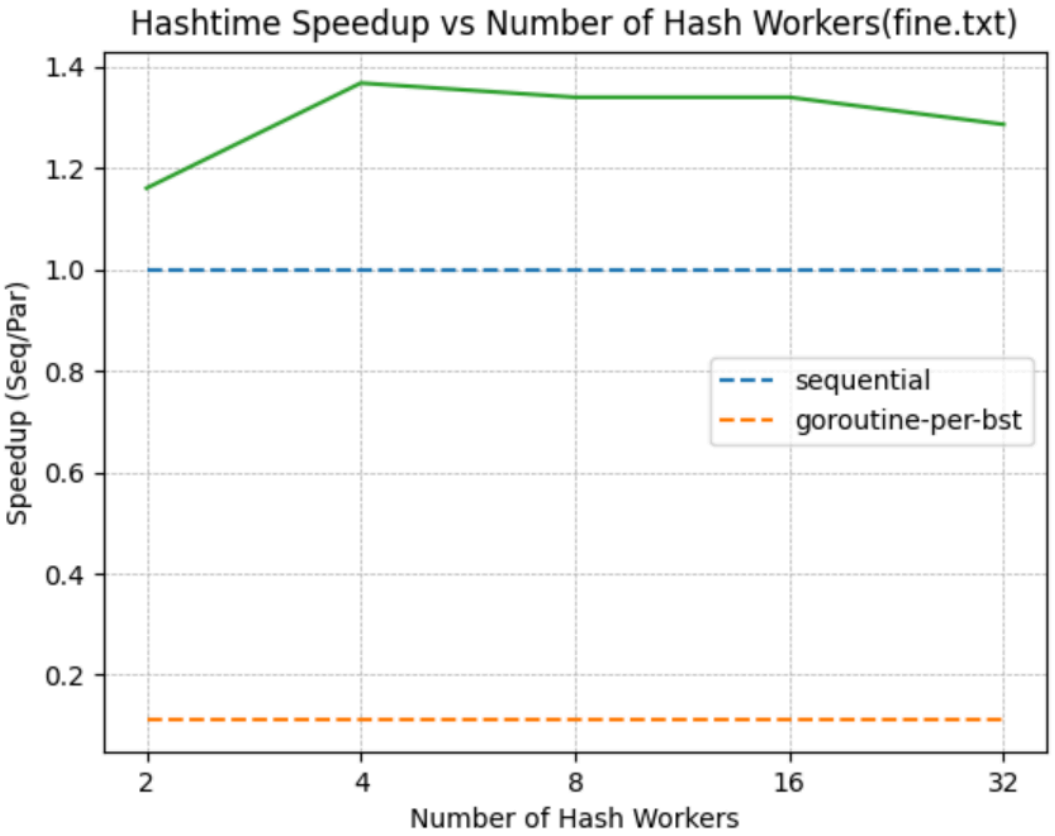

Figure 1b. Hashtime Speedup(fine.txt)

In coarse.txt, both implementations of spawning gr per bst and hashworker gr performed similarly. Spawning gr per bst slightly beats hashworker gr, however, as shown in figure 1a, hashworker gr slightly beats gr per bst at 8 threads. In fine.txt, gr per bst performed significantly worse than hashworker gr. When spawning grs, it is important to consider the granularity of work it will perform. For coarse work, we can worry less about how many threads to spawn because a large part of the gr runtime will be dominated by work instead of overhead. In Figure 1a, the speedup peaks at 8 grs and declines slowly from there, indicating an increase in overhead and a need to be cautious about spawning too many grs. For fine work, gr per bst has issues performing well against either hashworker or sequential due to a decrease in the amount of work performed for the increased overhead. Thus, we must be cautious about spawning too many grs when work granularity is fine.

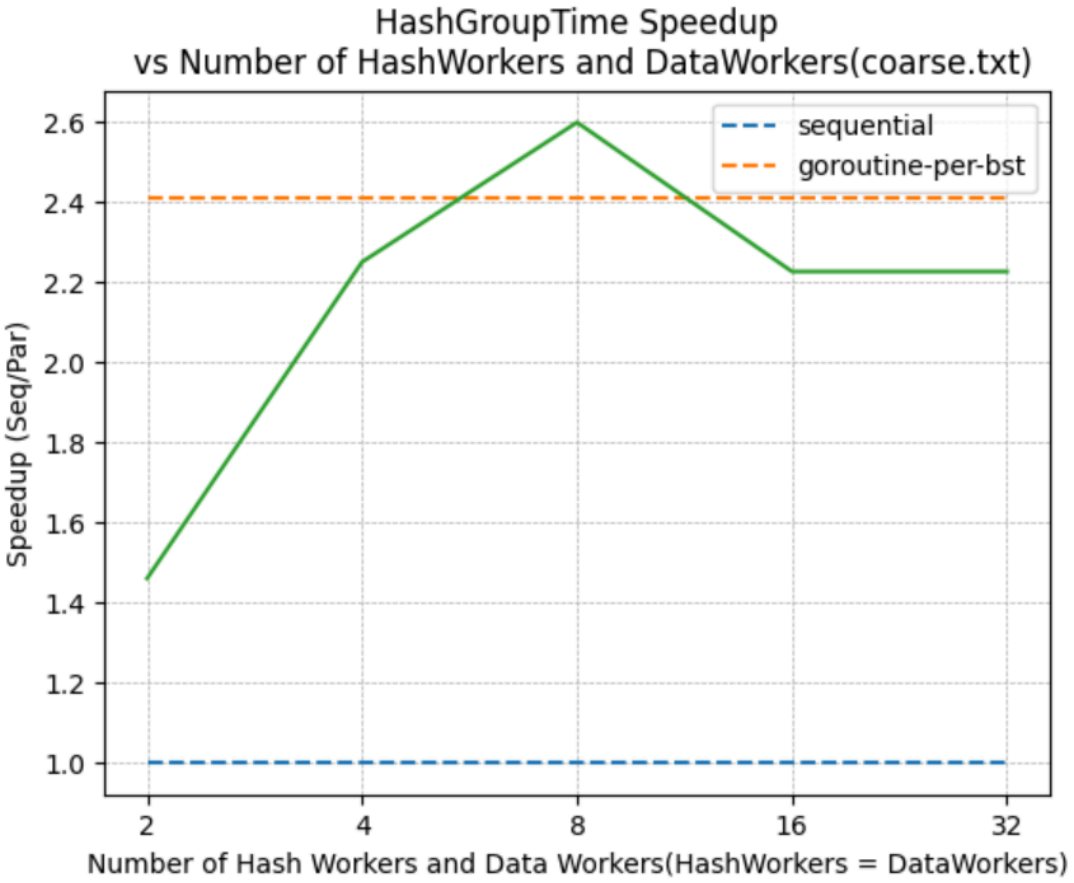

Figure 2a. HashGroupTime Speedup(coarse.txt)

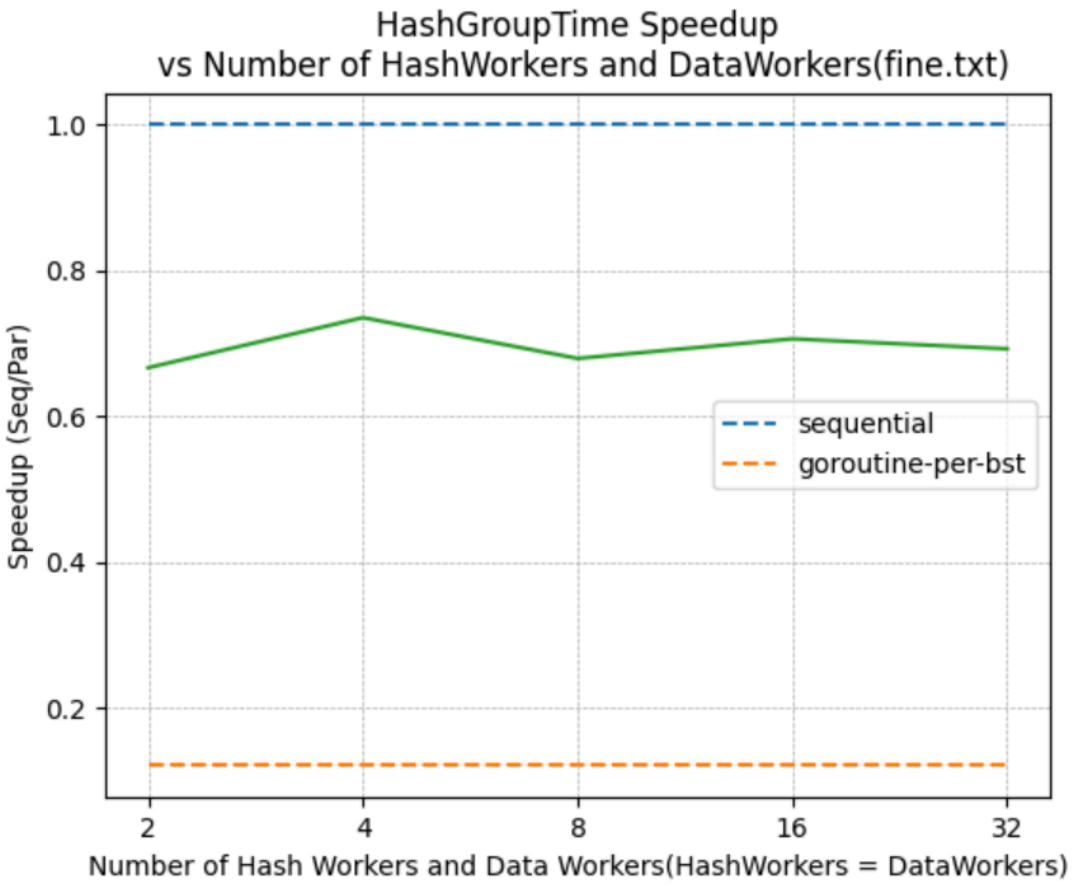

Figure 2b. HashGroupTime Speedup(fine.txt)

For hashGroupTime in coarse.txt, there is a similar speedup pattern because a large portion of the runtime is dedicated to hashing. In fine.txt, the overhead of spawning grs to update data seems to perform poorly against sequential. Because storing data into an array is not a costly operation, gr per bst incurs more overhead. When looking at raw runtimes for coarse.txt, there are no observed differences within e-10. However, looking at fine.txt, there is more overhead than work done, which explains the lower speedup. I found the sequential implementation easiest to understand, and, out of the parallel implementations, gr per bst was easier to understand because my implementation did not have race conditions, whereas my dataworker implementation did.

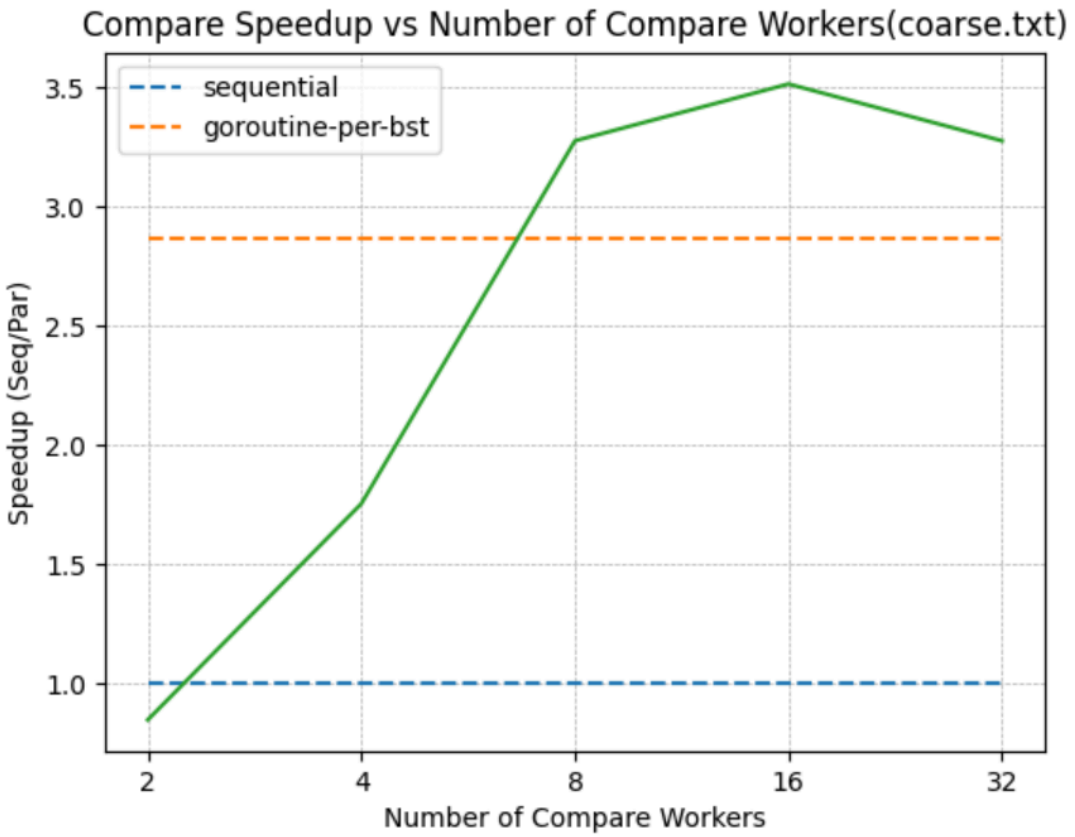

Figure 3a. CompareTreeTime Speedup(coarse.txt)

As shown in Figure 3a, with a low number of compareWorkers, it performs worse than gr per bst. As the number of workers increases, it ends up performing significantly better than gr per bst. They are similar in complexity, however, using compareWorkers is slightly harder to implement due the use of mutexes in my implementation. They both perform significantly better than a single thread because there is a lot of work compared to the overhead incurred. Depending on the workload and use case, using thread pools may be worthwhile to squeeze out more performance, but spawning a gr per bst has been shown to perform similarly to its more complex alternative.