10/20/2023

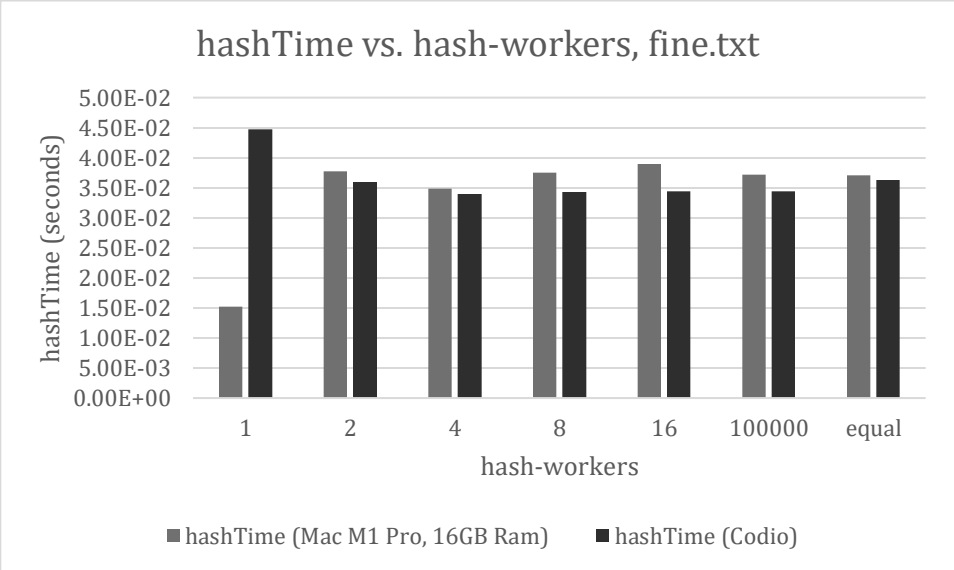
LAB3 – CS 380P – JOHN PETERSON - JLP5729

1. CREATE A SEQUENTIAL SOLUTION

My sequential program uses a map for hashGroups and compareGroups; no channels or other “fancy” Golang features were required. However, the various packages like flag and data structures like Slice made it much more simple than the last two labs in C/C++.

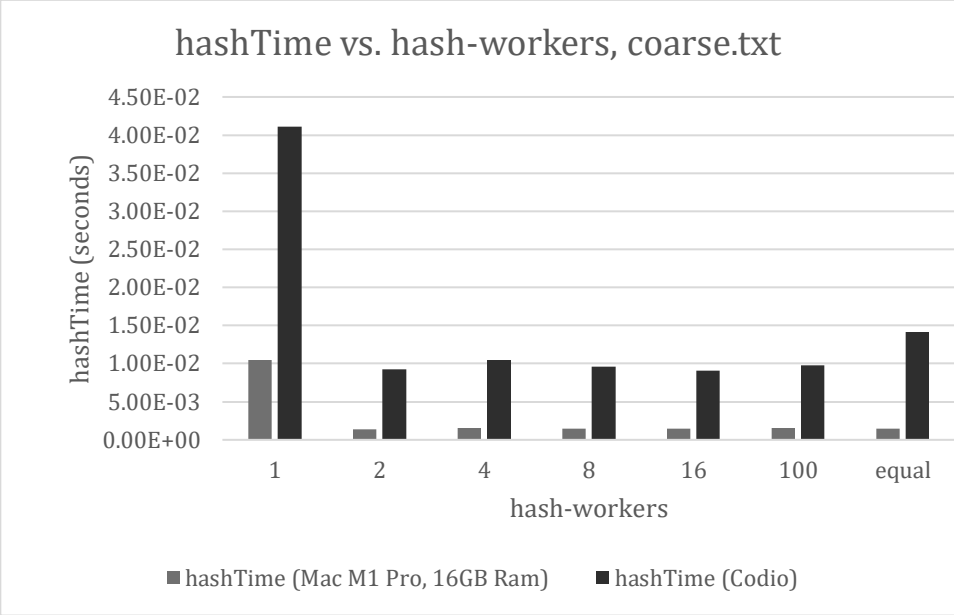
2. PARALLELIZE HASH OPERATIONS

The first implementation of parallel is spawning a goroutine for each BST. When operating on the “coarse.txt” file, compared to sequential, it achieved a 75% speedup on Codio and an 86% speedup on my M1 Pro Macbook Pro with 16GB of RAM (hence referred to as Macbook). However, on the “fine.txt” file, this achieved only a 29% speedup over running sequentially on Codio and actually incurred a 150% slowdown on my M1 Pro Macbook Pro with 16GB of RAM. This might be due to a less parallelized, mobile-class processor or less efficient implementation of Golang on the Apple ARM64 architecture and the overhead of setting up and tearing down goroutines on a less optimized or parallelizable architecture.



In fact, just spawning the number of goroutines equal to number of trees was roughly comparable to launching different number of goroutines and iterating; it was within 15-20% margin of performance for various hash-worker values (2,4,8,16) for fine.txt and in fact, some of the hash-worker values showed faster performance. It does appear that

there is a balance to strike between parallelization and performance (as demonstrated previously in this course).



I decided to not "sweat" the number of hash calculation workers when working on the next parts of Part 2 for hash compare since results were similar. If I were targeting a specific platform that had worse parallelization, I might want to limit the number of goroutines launched.

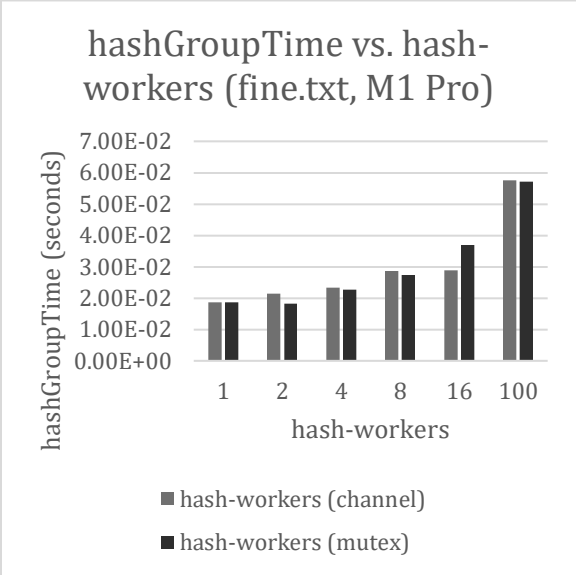
I expected going into this assignment and the hashTime section that Golang would handle threads efficiently enough that spawning one goroutine for each operation necessary in parallel would either be most efficient, or rather comparably efficient to more complicated options due to the concurrent nature of Go. I was surprised to see that there was a happy middle-ground where a certain number of goroutines would be most efficient. It was also interesting to see difference on hardware, comparing between the ARM64 platform and an Intel-based Codio machine.

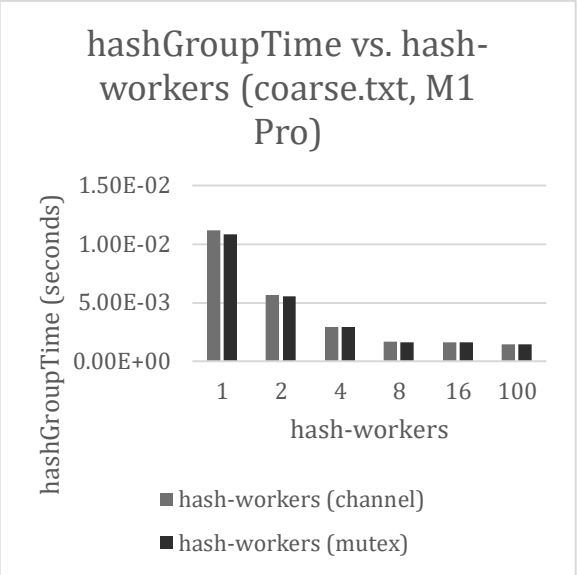
Following this, I implemented both the channel-based and mutex-based options for hash comparison. I saw a slight performance benefit to the channel-based option both on fine.txt and coarse.txt, as well as on both the Macbook and Codio. However, as the number of hash-workers increased, the mutex did worse (predictably so) as more threads led to more contention for the lock.

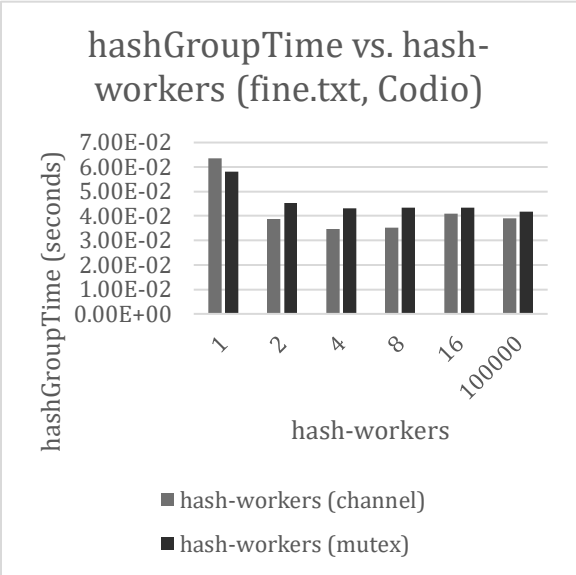
I did not attempt the fine-grain synchronization, but I imagine this might alleviate some lock contention and could result in a fine-grained structure winning out over a channel.

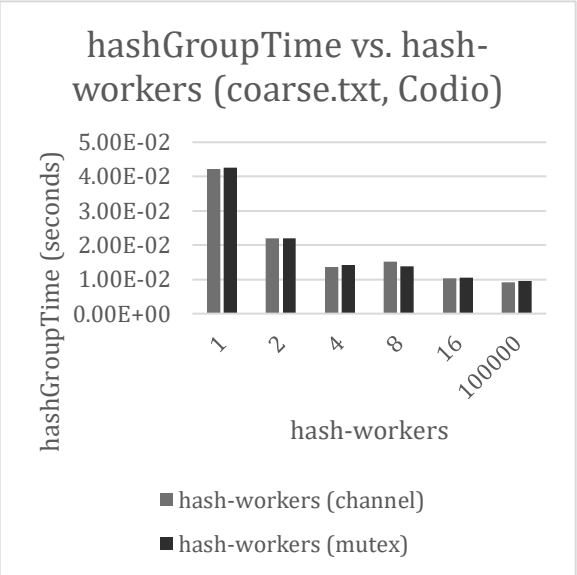
For my project, I decided that the channel-based method was more efficient than basic mutex, with about a 5-15% speedup across most hash-worker values on fine.txt but was pretty similar on coarse.txt. It stands to reason that since channels are a much

more commonly used synchronization device in Golang it might have better support and a more efficient implementation. See the graphs below for performance details on









mutex vs. channel. In either case, they both demonstrated a significant speedup over sequential. Channels showed less overhead than mutex on Codio but slightly more overhead on the M1 Pro Macbook. This again speaks to taking into consideration target hardware for optimizations.

I actually found the channel implementation simpler, as this felt more natural to write in Golang. For hashGroupTime, I expected much higher speedup from the channel method due to it being a feature of Go. One Mutex for thousands of threads seemed like it wouldn't work well at all. In fact, the results were much more mixed. The channel method largely did better than the mutex method, but only by a small amount and the

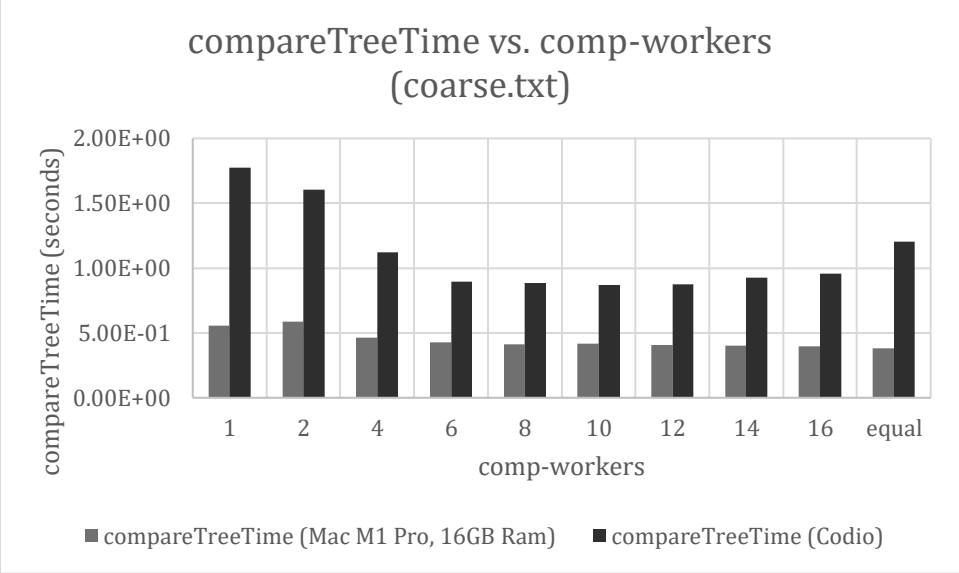
Codio machine actually performed better with the Mutex on coarse.txt (where more computation was required).

3. PARALLELIZE TREE COMPARISONS

For this section, we were asked to implement an adjacency matrix for tree comparison. The initial approach was to spawn one goroutine for each required tree comparison (when there were two or more trees with the same hash).

I observed approximately a 30% speedup on both the Macbook and Codio over sequential when implementing the Adjacency Matrix and launching a goroutine each time a comparison was necessary.

Implementing the buffer, I ran into challenges including deadlocking on condition variables (Wait only being able to be exited with a signal or broadcast, specifically). Once I determined that I could push a "dummy" value or create a Close function to check, atomically, if the thread was still "awake," I could then broadcast notEmpty to get the deadlocked routines to finish.



As you can see in the graph above, I had a slight speedup on both the Macbook and Codio when implementing comp-workers in the buffer format. However, the Macbook achieved the greatest speedup on the "equal" method, or launching a goroutine for each comparison. Codio machine achieved the greatest speedup between 10-12 comp-workers, achieving nearly a 51% speedup. However, and this may be due to the inefficient nature of my code (Broadcasting many times which could cause threads to wake up unnecessarily), as we approached higher thread count/comp-workers, Codio machine saw higher overhead.

I think managing a thread pool could definitely be worthwhile, especially when working with something with large computational requirements. Additionally, the assignment

required us to use a custom buffer, but Go allows for buffered channels, and that would make managing a thread pool nearly trivial. I think using buffered channels and a worker pool would strike the perfect balance between performance and ease of use.

I expected for compareTreeTime that the buffer would be the fastest, as it wouldn't be worthwhile coding one unless we would get some speedup. However, I was surprised to see that it wasn't as stark of a speedup on the Macbook, or even on the Codio, compared to just launching one goroutine for each comparison. It was also interesting to see the overhead in my Buffer implementation causing the Codio time to rise as comp-workers increased past 10-12. It is especially interesting because previous implementations of "just launch all the goroutines you could need" didn't provide the best speedup on the Macbook, but it did for this section. I think this likely has to do with the implementation of the condition variables and sync functions I utilized on the Mac architecture/software.

Either way, it was an excellent learning experience.