10/20/2023

Lab3 – CS 380p – John Peterson - JLP5729

# 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++.

# 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.

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.

# 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).