Go BST Comparison

Talha Khalil

October 25, 2022

Contents

List of Figures

1 Introduction

The goal of this lab was to gain insight into writing a program in a programming language (Google's Go) which restricts the programming model to simplify dealing with concurrency. Through this lab we will find identical Binary Search Trees (BSTs) by computing the hash of the BST, as well as comparing the BSTs through inorder traversal if the hash is identical. We make use of Go's Concurrency features, as well as write our own to have a reference to compare with.

1.1 Hardware

A note to be made about the hardware, is that all the plots shown were run on Codio. All the collected data points were also obtained on Codio. Another note about speed differences here is that due to the high usage of the Codio environment on the day of the testing, some numbers were more elevated than prior testing, specifically for runs which required more memory.

1.2 Plots

All data points taken are an average of 10 runs back-to-back. This was chosen as this is also what the auto grader will be doing.

The total time spent on this lab was approximately *40 hours*.

2 N Goroutines vs Hashworker Threads

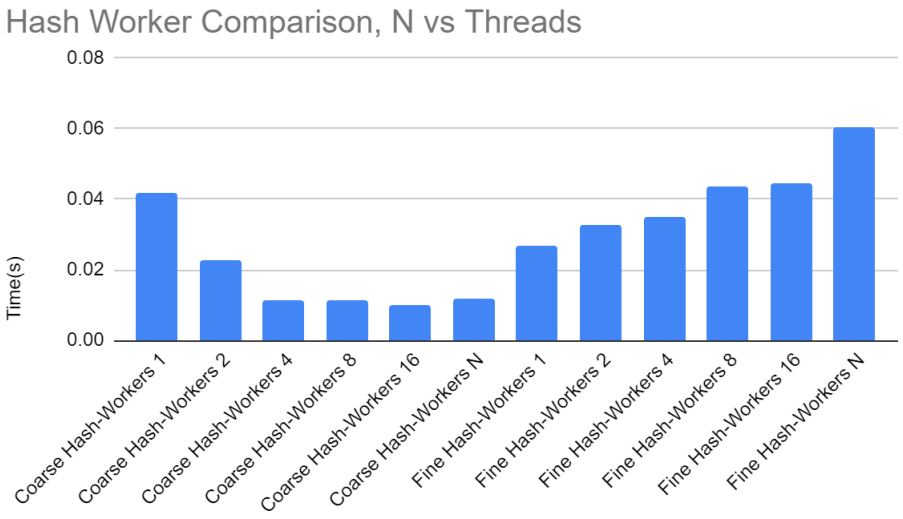

Figure 1: N Goroutines vs Hash Worker Goroutines

When comparing the amount of goroutines launched for hashing, a clear trend can be seen for both Coarse and Fine inputs. However, the trend is not the same between them. With the Coarse input file, a file containing a 100 BSTs with large amounts of nodes, we can observe a speed up when the amount of goroutines launched increases. This trend does stop when we launch N (100) goroutines which we can see is slower than 16. From this we can conclude that there is a limit to how many goroutines Go can manage. Showing that there are limits to the performance benefits it can bring, and could rather be more harmful if more goroutines are launched.

When looking at the Fine input file it can be observed that adding more routines causes a slow down. This can be attributed to the fact that the BSTs are much smaller than of Coarse, and parallelizing the action adds more overhead, than just computing the hash sequentially.

Even though Go can manage goroutines for us, one should still be cautious in how many routines are being launched as it is clear that depending on the work being parallelized it could be detrimental rather than helpful.

3 Mutex vs Channels

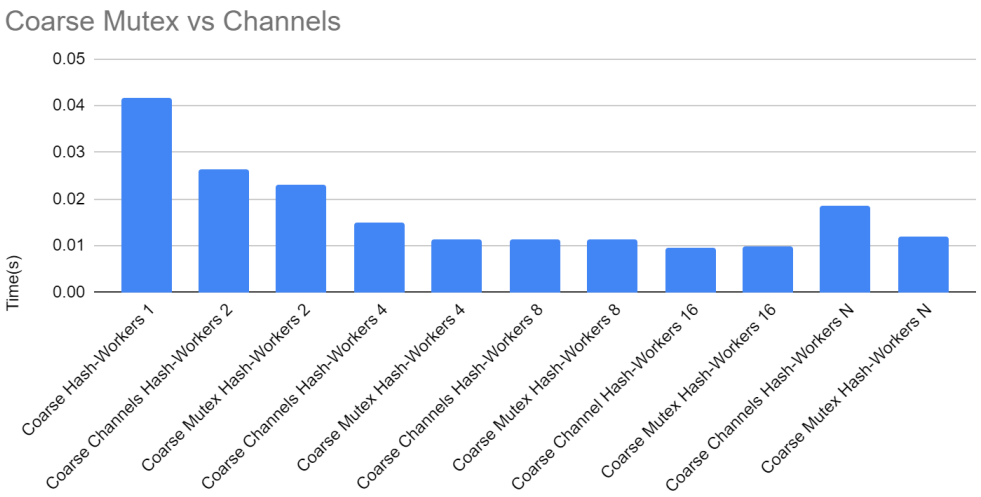

Figure 2: Coarse Mutex vs Channel Hashing

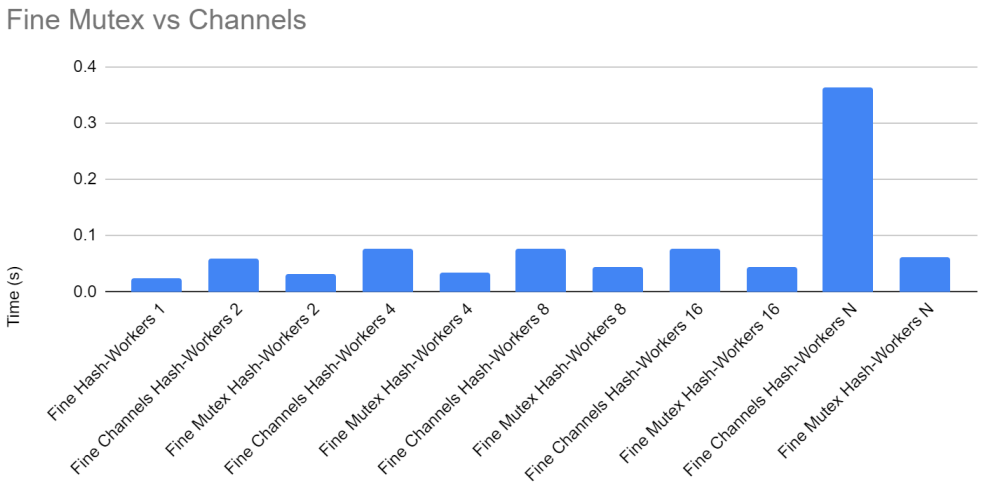

Figure 3: Fine Mutex vs Channel Hashing

When comparing the usage of Mutexes, vs Channels to manage shared memory, from the above figures we can definitely see a clear trend between both input files. Where it is clear that Channels add more overhead than good usage of

Mutexes when managing shared Memory. As discussed previously the Fine input text does not benefit from parallelization and actually performs worse, and we can observe that with N Channels it is much worse than N Mutex locks. From 2 it is apparent that when the work being parallelized takes long enough the usage between channels vs mutexes is negligible. Because of which it is apparent the programmer make use of the simpler and safer implementation, which we believe is Channels.

A special note about the implementation used here for this lab is that the hashing and hashing and inserting is closely tied together, so their timing is identical.

4 N Goroutines vs Thread Pool

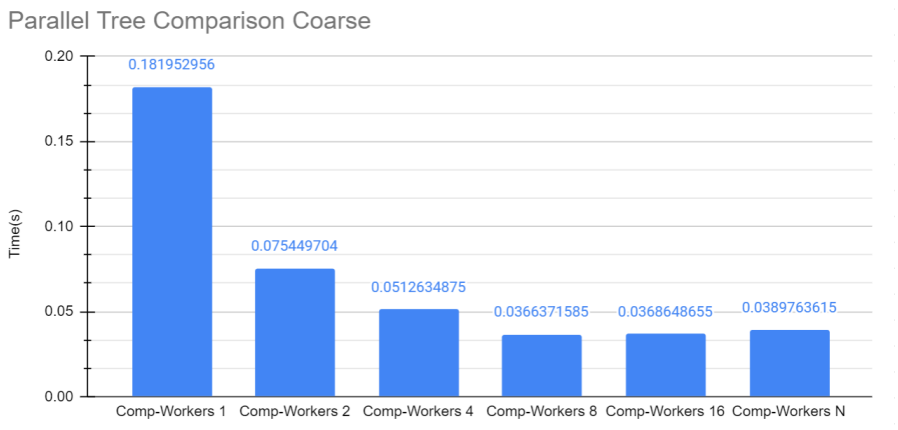

Figure 4: Tree Comparison Thread Pools vs Goroutines for Coarse

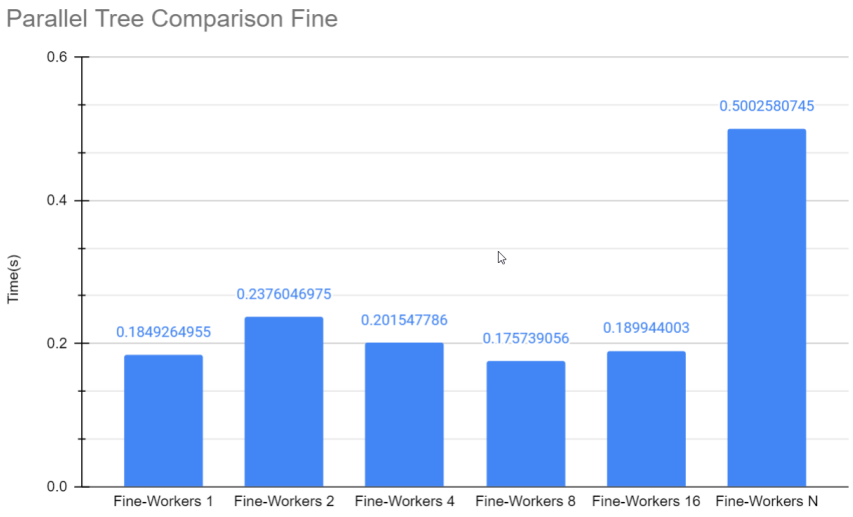

Figure 5: Tree Comparison Thread Pools vs Goroutines for Fine

We observe similar behavior as we did with the last section, where it is very important what operation is being parallelized. Where we can clearly see that Fine text did not have improvements when increasing the number of workers. However, the story continues with the speed up observed in Coarse where increasing the goroutines caused a direct increase in performance. We can also see that managing our own thread pool yields similar results to letting Go handle it. This clearly suggests that managing our own thread pool is not worthwhile, as the goroutine implementation was much easier and faster to

implement and yields similar performance.

5 Summary

In summary, we found that Go can be a powerful language when used correctly, and can cut down on development time as it has good error messages built in. As well as a deadlock identifier making it easier to program concurrent programs. It would definitely be beneficial to use Go in the future for writing performant applications especially if one does not need to worry about interfacing with legacy C/C++ code as it is much easier to write concurrent programs within it. One key take away is that like always, one should be careful about what they are trying to parallelize as some application cases would rather decrement in performance if parallelized due to the overhead.

5.1 Insight Gained

Some of the key insight gained from doing this lab is that string concatenation is very costly. We were using string concatenation recursively which caused a huge bottleneck, which could be easily observed in parallelizing tree comparisons where it used to take minimum of 5 seconds, to compute. Where as with string concatenation handled it takes milliseconds.