BST with GO - Lab #3 Report

Abraham Zamora

10/28/2024

1 Introduction

This lab aimed to explore the effectiveness of parallelization in Go by creating binary search trees and then determining binary search tree (BST) equivalence. By leveraging goroutines, channels, and synchronization techniques, I aimed to achieve speedup over a sequential implementation. The experiment was conducted on two main datasets (`coarse.txt` and `fine.txt`) to evaluate scalability and efficiency. `Coarse.txt` had a smaller number of BSTs, but the BSTs contained numerous nodes. `Fine.txt` had a large number of trees (100,000) but the trees were particularly small. As usual, I used speedup values $\left(\frac{T_{\text{seq}}}{T_{\text{parallel}}}\right)$ to standardize our results and highlight the impact of different parallel configurations.

2 Step 1: Sequential Solution (Baseline)

The sequential solution served as our baseline, constructing BSTs from input files, generating hashes through in-order traversal, grouping trees together with similar hashes, and comparing trees with matching hashes to confirm equivalence. The time values from the sequential implementation serve as T_{seq} for calculating speedups in subsequent steps.

3 Step 2: Parallelizing Hash Operations

3.1 Implementation Approaches

In order to determine the speedup for a parallel implementation and to test the benefits of using parallelization in Go, I implemented a few different approaches for hashing and grouping the trees that have similar hashes. The implementations are as follows:

1. **Hash-Workers Only**: Employed a fixed number of hash workers to compute hashes concurrently.

2. **Hash and Data-Workers**: Added data workers to manage shared access to the hash storage map, using either channels or locks to ensure thread safety.

3. **Optional Fine-Grained Sync**: Allowed multiple data-workers to access the shared map simultaneously, with fine-grained locks ensuring that only one thread could modify an entry at a time.

For each configuration, the number of hash-workers and data-workers depended on the input parameters. By default, the program runs sequentially with one hash-worker and one data-worker. Additionally, setting hash-workers to zero runs a configuration with a single Go routine for hashing.

3.2 Results and Speedup Calculations

Using Speedup $= \frac{T_{\text{seq}}}{T_{\text{parallel}}}$, we calculate speedup values for `fine.txt` and `coarse.txt` with each configuration:

Configuration	**Fine.txt Speedup**	**Coarse.txt Speedup**
Sequential	1.00	1.00
One Go Subroutine	0.92	2.10
Hash Workers Only		
`2`	1.50	2.18
`4`	1.96	3.71
`6`	2.32	3.40
`8`	1.71	2.88
`16`	1.68	2.88
`N`	0.20	2.78
Hash and Data Workers		
`2`	1.60	2.47
`4`	1.77	2.26
`6`	2.95	2.82
`8`	2.52	3.28
`16`	2.44	3.05
Optional Implementation		
`4:2`		3.72
`6:4`		2.88
`8:6`		2.33
`16:8`		3.15
`N:16`		4.01

Table 1: Speedup for Hash Operations

3.3 Discussion

- **Overhead and Complexity**: The single-lock approach introduced noticeable overhead due to contention, especially with high worker counts. The channel-based approach had less contention and offered better performance up to moderate worker counts. This can be seen in the Coarse sample file, as it had better performance over the sequential when using the channel based approach. The lock approach worked well with the Fine.txt file when the data-workers increased to 4 or more, suggesting that using lock based approaches can work when N increases greatly. Achieving 2.5 times speedup would definitely be noticeable as N continues to grow.

- **Speedup Compared to Sequential**: The single-channel solution was simpler and performed well with moderate concurrency, while fine-grained synchronization using multiple data workers achieved the best overall speedup. This is particularly true with the Coarse.txt file. Using the optional implementation proved useful here for serious speedup.

- **Bottlenecks**: Access to the shared data structure was a bottleneck in high worker configurations, particularly with N workers. The fine-grained approach alleviated this bottleneck by allowing controlled parallelism in data access. However, it should be noted that this was not the case for the fine.txt file. It appeared that the optional implementation had too many bottlenecks with fine.txt and never provided a reliable measurement.

4 Step 3: Parallelizing Tree Comparisons

4.1 Implementation Approaches

The final step in the assignment was to run the tree comparison in parallel. For tree comparisons, the following configurations were used:

1. **Sequential**: This was the original sequential approach.

2. **Single Goroutine**: This approach used a single goroutine for comparison tasks, benefiting from Go's concurrency model without added worker overhead.

3. **Multiple Comparison Workers**: Workers processed tree comparisons through a concurrent buffer protected by mutexes and condition variables, ensuring that tasks were non-blocking and non-spinning. The number of comparison workers ranged from 2 to 16 based on the assignment guidelines.

4.2 Results and Speedup Calculations

The results for each of the implementations can be seen below.

Configuration	Comparison Speedup to Sequential
Single Go routine	2.42
Sequential	1.00
`2 comp-workers`	3.52
`4 comp-workers`	3.64
`6 comp-workers`	3.2
`8 comp-workers`	3.42
`16 comp-workers`	3.05

Table 2: Speedup for Tree Comparisons

4.3 Discussion

- **Performance vs. Complexity**: The single-goroutine approach provided a straightforward solution with a 2.42x speedup. The added workers improved upon the intial times, achieving over 3x improvement across. The best performance was at 4 comp-workers. With 4 comp-workers, we add some overhead, but provide a good balance between overhead management and performance. Additional comp-workers did not seem to provide much, if any benefit.
- **Scaling and Efficiency**: With balanced concurrency, we can achieve noticeable speed up. The best performance was with 2 or 4 comp-workers, providing a balance between the overhead and good performance.
- **Thread Pool Management**: The complexity of managing a thread pool was worthwhile up to 4 workers, but beyond that, the returns diminished. There was some improvement at 8 over 4, but it could also just be my how Codio computer managed the threads.

5 Conclusion

This lab helped to demonstrated the effectiveness of parallelism in Go for BST equivalence. Moderate worker counts consistently yielded the best speedup, which makes intuitive sense. The fine grained approach worked particularly well with Coarse.txt, achieving significant speedups compared to the channel approach and the lock approach that had been used. In fact, when N equaled the number of trees and there were 16 dataworkers, the implementation had a 4x speedup, which is substantial. The lab overall demonstrated the importance of balancing concurrency and synchronization costs. Well-managed parallelism can outperform sequential implementations in complex tasks.

Finally, in terms of the time this lab took, it was a bit less time than the previous assignment. Overall, the coding and testing of the lab took around 16 hours, with an additional 2 for writing out this report. Thus, in total, this assignment took around 18 hours total. The lab was largely completed on Codio, and I only transferred the program over to my computer to run the fine.txt file,

as the large number of trees would print out better for comparison with other solutions (it was mentioned in Ed Discussion that students can compare the outputs of the test runs, which I compared my output of fine.txt with those posted on Ed Discussion). Once I compared the solutions, I moved back to finish on Codio.