BST Comparison Performance Report

Siddharth Sundaram

February 12, 2025

1 Introduction

1.1 Abstract

In this lab, I wrote a Go program to determine the equivalence of Binary Search Trees (BSTs). The implemented algorithm to accomplish this consists of hashing each BST, grouping BSTs with the same hash, and only comparing BST pairs that have the same hash. For two BSTs to be equivalent, they must have the same in-order traversal, and the hash function involves an in-order traversal of the BST. This report details the performance benefits of different parallel implementations when compared to the sequential implementation with empirical data to support claims.

1.2 Algorithm, Implementations, and Data Structures

At first, I implemented a sequential approach that used only the main thread to perform the algorithm. After verifying the sequential approach, I parallelized different parts of the algorithm that could benefit from concurrency.

The hash computation phase was parallelized using two different implementations: one creates a goroutine per BST, so each goroutine computes the hash of exactly one BST, and the other creates a fixed number of goroutines specified by the flag hash-workers and divides the BST hash computations among them.

The hash grouping phase was also parallelized using two different implementations: one creates a ”central manager” goroutine that is responsible for putting BST IDs in a map where the BST hash is the key, and the other has the goroutines that computed the hashes contend for a lock to populate the map. In the first implementation, the goroutines that compute BST hashes send the (hash, BST ID) pairs to the central manager goroutine through a channel. In the second implementation, the goroutines attempt to acquire a mutex after computing a BST hash, and only after succeeding do they add the (hash, BST ID) pair to the map.

Finally, the tree comparison phase was also parallelized using two different implementations: one creates a goroutine per BST pair, so each goroutine compares exactly one pair of BSTs, and the other creates a fixed number of goroutines specified by the flag comp-workers and divides the BST comparisons among them. In the second implementation, the main thread enqueues BST pairs to a concurrent buffer, and the comp-workers goroutines dequeue pairs from this buffer and compare them.

The concurrent buffer's underlying data structure is a linked list with head and tail pointers, so that enqueueing and dequeueing from the buffer is O(1). Additionally, the buffer has a condition variable for enqueueing and a condition variable for dequeueing, along with a mutex to protect the shared data structure. When the main thread tries to enqueue a BST pair, it is blocked if the buffer has reached capacity and will only be unblocked when the buffer is no longer at full capacity. After the main thread successfully enqueues a BST pair, it signals a comp-worker goroutine to wake up and dequeue a BST pair to compare. Similarly, when a comp-worker goroutine tries to dequeue a BST pair, it is blocked if the buffer is empty and will only be unblocked when the buffer is not empty. After a comp-worker thread successfully dequeues a BST pair, it signals the main thread to wake up and enqueue a BST pair. Finally, when the main thread has enqueued all BST pairs to be compared, it wakes all blocked comp-worker goroutines with a broadcast so they can finish execution. Each interaction where the buffer is being modified is guaranteed to be a critical section because it is protected by the mutex and condition variables.

2 BST Hashing Grouping Analysis

2.1 Hash Computation

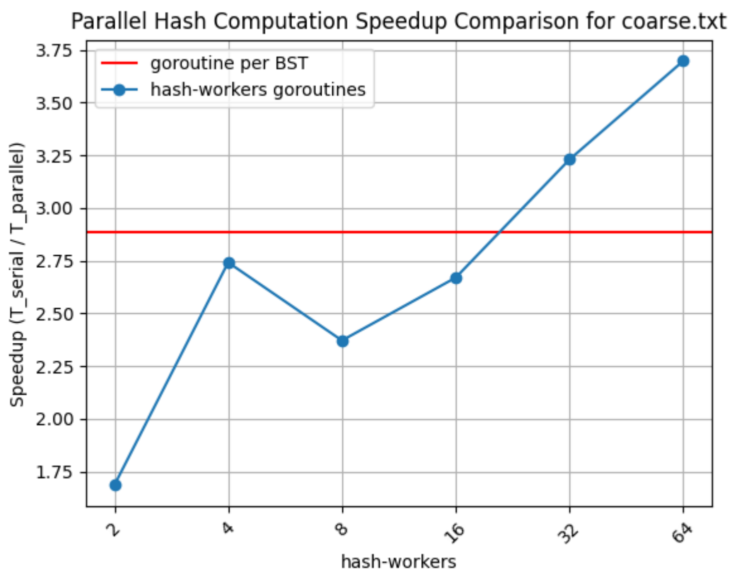

Figure 1: Speedup comparison of parallel hash computation implementations on coarse.txt

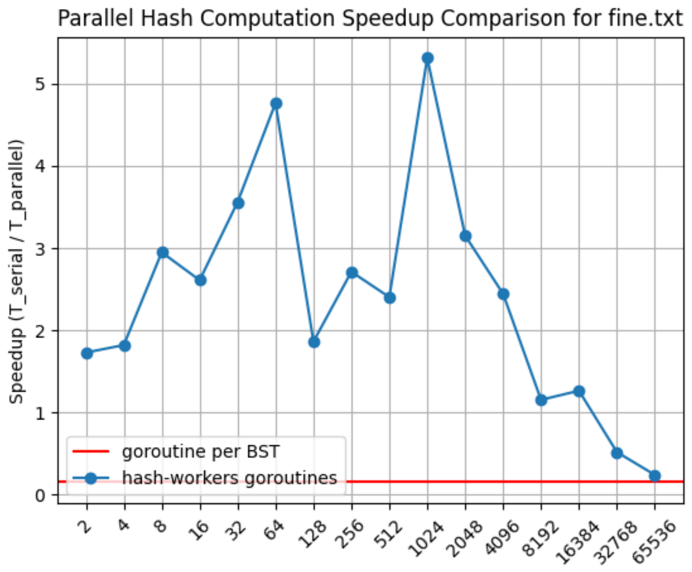

Figure 2: Speedup comparison of parallel hash computation implementations on fine.txt

As seen in Figure 1 and Figure 2, which parallel hash computation implementation is faster depends on the input and the number of hash-workers chosen for the implementation that requires it. Figure 1 suggests that creating a goroutine per BST is more performant than smaller numbers of goroutines for small input sizes where the trees themselves are large (coarse). That being said, the hash-workers goroutines implementation outperforms the goroutine per BST implementation when hash-workers is greater than 16. Figure 2 shows that the hash-workers goroutines implementation is consistently faster than the goroutine per BST implementation for all values of hash-workers that were tested. This suggests that there is some fixed number of goroutines (somewhere around 1024 in the case of fine) that performs best when there are a very large amount of BSTs and the trees themselves are small (fine). While both parallel implementations performed better than the sequential implementation for coarse, only the hash-workers goroutines implementation performed better than the sequential implementation for fine. For this reason, the hash-workers goroutines implementation is overall more performant than the gouroutine per BST implementation.

Even though there was a lot of variation in speedup for different numbers of goroutines, it seems that Go manages goroutines well enough that the user usually doesn't have to worry about how many threads to spawn. There does seem to be an ideal number of goroutines that results in the best performance, but the range is very large and most values resulted in speedup. That being said, if the user is thinking about spawning millions of goroutines, they should proceed with caution, because the lightweight nature of a goroutine is less evident when there are millions of goroutines that Go needs to manage.

2.2 Hash Grouping

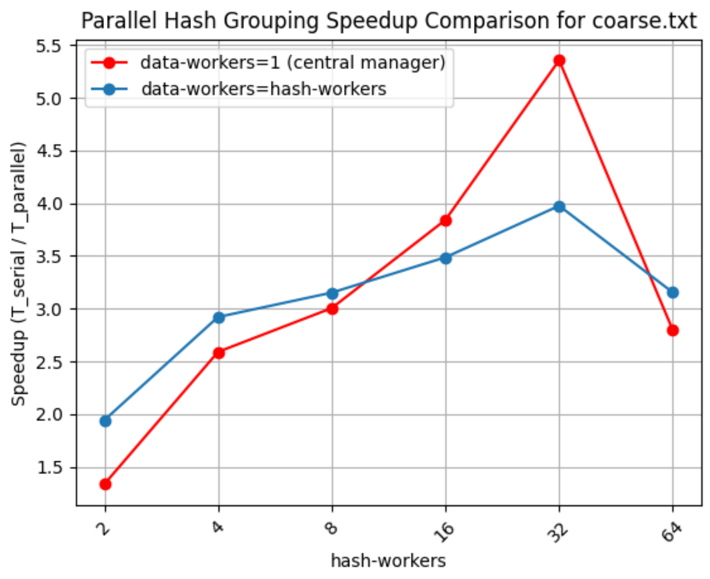

Figure 3: Speedup comparison of parallel hash grouping implementations on coarse.txt

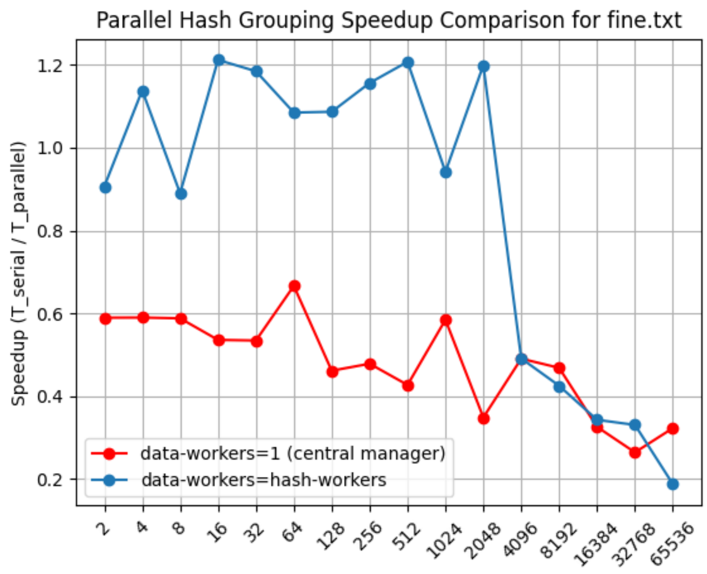

Figure 4: Speedup comparison of parallel hash grouping implementations on fine.txt

According to Figure 3, both implementations of parallel hash grouping perform and scale similarly with the number of hash-workers goroutines. It seems that the central manager implementation significantly outperforms the data-workers=hash-workers implementation when the number of hash-workers is 32, but the overall trends of both implementations are very similar to each other and both perform better than the sequential implementation. According to Figure 4, the data-workers=hash-workers implementation outperforms the central manager implementation for many values of hash-workers (2-2048), and performs slightly better than the sequential implementation when hash-workers is between 16 and 2048. The central manager implementation consistently performs worse than the sequential implementation for all values of hash-workers tested. Both implementations have similar performance at larger values of hash-workers (4096-65536). The figures suggest that, with the ideal number of hash-workers, the central manager implementation is more performant for inputs with a small number of large trees (coarse), while the data-workers=hash-workers implementation is more performant for inputs with a large number of small trees (fine).

Based on the results, it seems like the central manager implementation has more overhead than the data-workers=hash-workers implementation. The former uses a channel to achieve communication between goroutines, which means that sending goroutines must acquire a mutex in order to put their data into the channel, and the receiving central manager goroutine must acquire the lock to take data out of the channel. This is two lock acquisitions for each time data is added to the map. The latter uses a mutex to access and modify the map, therefore serializing map population, and there is only one lock acquisition for each time data is added to the map. Since the former implementation has approximately double the lock acquisitions as the latter, it makes sense that it also has more overhead. Additionally, since the central manager goroutine is the only one with access to the map, map population is serialized in this implementation as well.

Regardless of performance, I find the central manager implementation to be a much simpler approach than the data-workers=hash-workers implementation because channels are inherently thread-safe, while mutexes require care and caution. Since the channel abstracts away the trickiness of parallelism, a user can make use of the channel without understanding synchronization problems. The same cannot be said about the use of mutexes.

3 BST Comparison Analysis

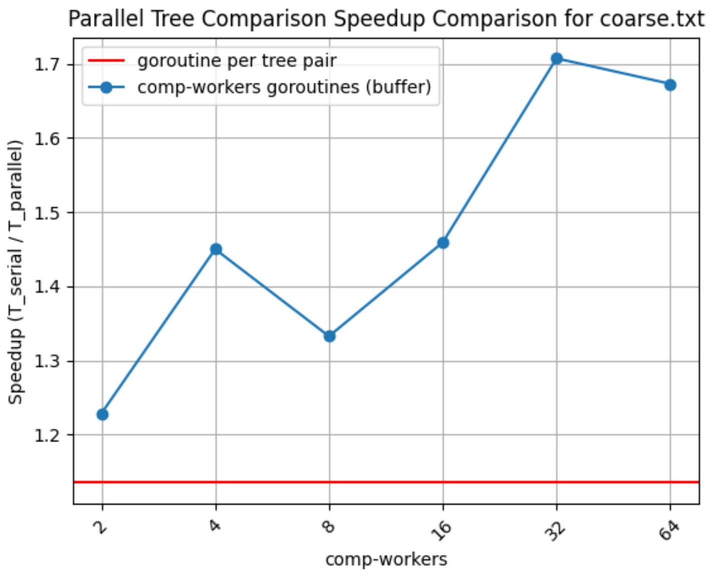

Figure 5: Speedup comparison of parallel BST comparison implementations on coarse.txt

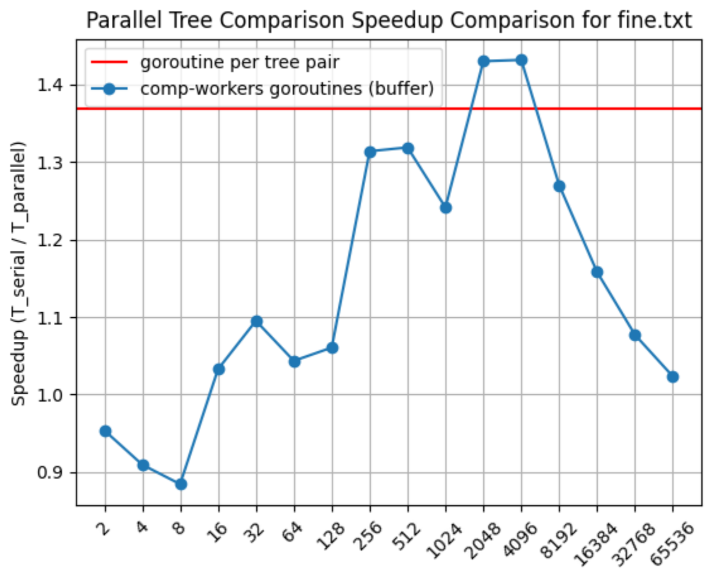

Figure 6: Speedup comparison of parallel BST comparison implementations on fine.txt

According to Figure 5, both parallel implementations performed better than the sequential implementation, but the buffer implementation consistently outperformed the goroutine per BST pair implementation. The buffer implementation seems to perform better with a larger number of comp-workers goroutines, but the ideal number of goroutines is likely between 32 and 64 (for coarse). The complexity of the per BST pair implementation is probably lower than that of the buffer implementation, since there is no lock contention due to each goroutine modifying a different index of the shared adjacency matrix. The complexity of the buffer implementation is probably higher because of lock contention when enqueueing and dequeueing BST pairs, but seems to be more performant due to a relatively low number of comp-workers goroutines. According to Figure 6, the goroutines per BST pair implementation outperformed the sequential implementation, and the buffer implementation outperformed the sequential implementation for values of comp-workers goroutines between 16 and 65536. For most values, the per BST pair implementation outperforms the buffer implementation, but the buffer implementation outperforms the per BST pair implementation for when comp-workers is between 2048 and 4096 (this is likely where the ideal number of comp-workers lies for fine). The difference in complexity is much more evident with fine, since there is much more lock contention with large values of comp-workers goroutines.

As seen in the figures, managing a thread pool with the buffer implementation barely performs better than the per BST pair implementation when the thread pool size is ideal. Given the very mediocre increase in speedup, however, it may not be worthwhile to go through the struggle of implementing the buffer and managing the thread pool, especially since it doesn't scale well with large numbers of comp-workers goroutines.