Nicolas Ayllon
EID: nja842

Lab 3 BST with Go

Introduction

In this lab, I wrote a program in the language Go to process binary search trees and place them into unique groups of equivalent trees. To save time processing large numbers of trees, the program first hashes the trees to identify potential duplicates, and only compares trees with the same hash. The steps are summarized below:

1. **Hashing**
   calculate a hash for each tree based on its inorder traversal
2. **Hash Groups**
   create a map from hashes → tree IDs
3. **Compare**
   compare trees in each hash group to find equivalent trees, and make unique groups
   Note: BSTs are equivalent if they have the same inorder traversal (contain the same values).

I tested the program on these inputs with different numbers of trees and nodes:

coarse.txt: N = 100 trees with 10,000 nodes (1,000,000 values total)

fine.txt: N = 100,000 trees with 10 nodes (1,000,000 values total)

After a sequential implementation, I used Go's concurrency features, such as **channels and goroutines** to parallelize steps in different ways, described in more detail in each section.

These command line arguments vary the implementation:

- `hash-workers`
  number of goroutines launched for hashing trees
- `data-workers`
  goroutines launched for making the map from hashes to tree IDs
- `comp-workers`
  goroutines launched for comparing trees within each hash group

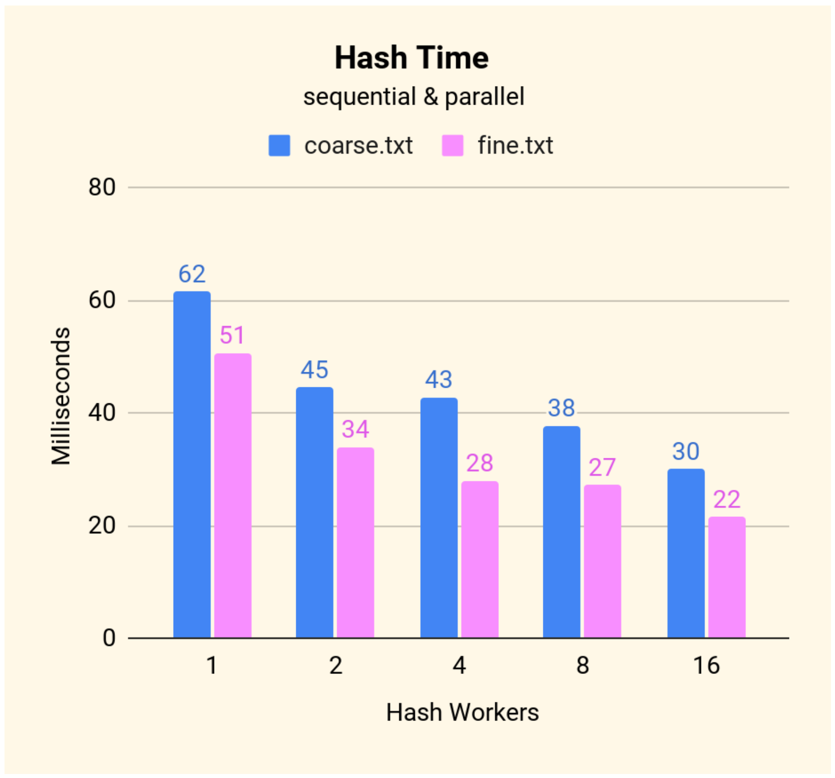
Step 1. Hash Trees

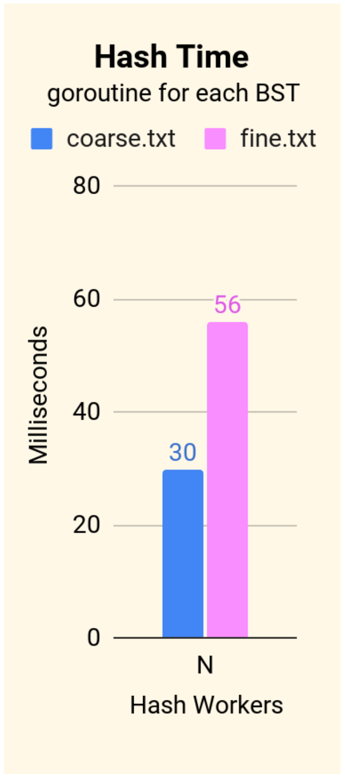
After implementing a sequential version of tree hashing, I parallelized this stage in two ways:

- Spawning a goroutine to hash each tree (N, the number of trees)
- Spawning `-hash-workers` goroutines to each hash a fraction of all BSTs

The following chart shows the results for an **average of *n* = 20** tests.

Nicolas Ayllon
EID: nja842





The data shows these results:

- The N goroutines version (one for each BST) saw mixed results compared to sequential:
  - In coarse.txt, execution time decreased: 74 ms → 33 ms
  - In fine.txt, execution time increased: 77 ms → 80 ms
- The `-hash-workers=2,4,8,16` version saw significant improvements with increasing hash workers compared to sequential:
  - In coarse.txt, execution time decreased: 74 ms → 36 ms (16 goroutines)
  - In fine.txt, execution time decreased: 77 ms → 34 ms (16 goroutines)
- Comparing both parallel versions:
  - The **N goroutines version runs faster on coarse.txt**. This might be because each of the 100 goroutines performs a significant amount of work hashing a large 10,000-node tree. The `-hash-workers=2,4,8,16` version decreases the execution time, but there's still some benefit to increasing goroutines to N = 100.
  - The **`-hash-workers=2,4,8,16` version runs faster on fine.txt**. With 16 goroutines, each processes 100,000/16 = 6,250 trees with 10 nodes each, giving each goroutine a significant amount of work. Increasing to N = 100,000 gives each routine very little work (hashing one 10-node tree), so the overhead negates any previous benefit to parallelism with fewer goroutines, and brings the execution time back up to 80 ms (close to the sequential time of 77 ms).

Nicolas Ayllon
EID: nja842

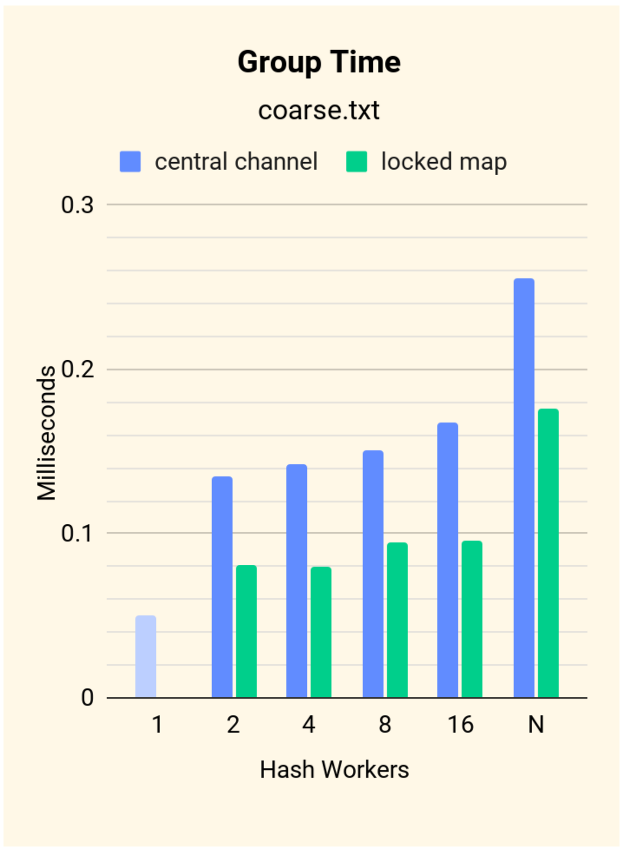
Evidently, using large numbers of goroutines does not always yield improvements, and may introduce overhead that counteracts earlier benefits to parallelism (100,000 goroutines on fine.txt). Although Go manages large numbers of goroutines well, **it's still worthwhile to choose a number of threads where each has enough work** to offset overhead.

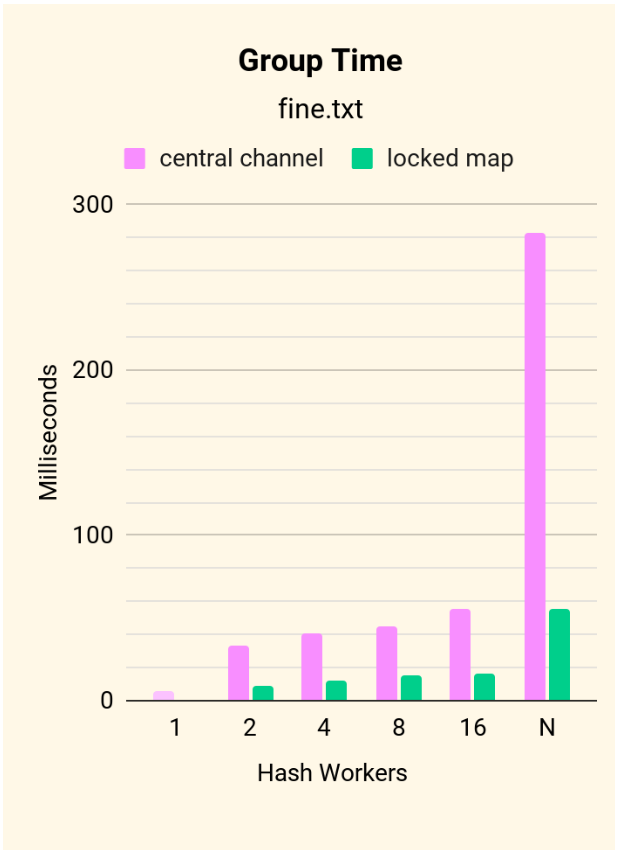
Step 2. Map Hashes → Tree IDs

After sequentially mapping hashes to tree IDs, I parallelized this step in two ways:

- **central channel**
  Hash workers send a (hash, tree ID) pair to a manager goroutine via 1 central channel, which inserts the pair into the map
- **locked map**
  Hash workers insert a (hash, tree ID) pair to a map protected by a single lock

I implemented Step 1 (hashing) completely separate from Step 2 (mapping), so the timing results below show **only the group time, not including hashing**.



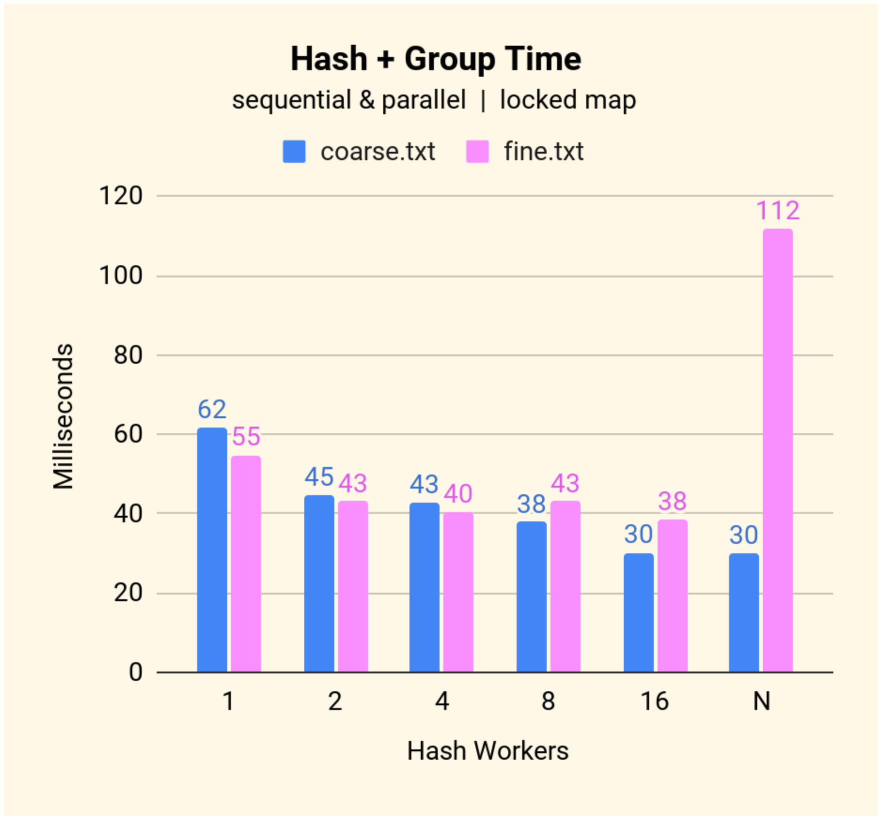


- After hashes are precomputed, the **sequential version makes the map faster** than either parallel version.
- The **central channel has more overhead**, since **locked map is faster** for both inputs:
  - In coarse.txt, locked map takes only **61%** the time of central channel (average)
  - In fine.txt, locked map takes only **29%** the time of central channel (average)

Nicolas Ayllon
EID: nja842

I found the channel version required less code to write than implementing the locked map, `singleLockMap`, but I found the locked map more intuitive than channels.

Considering **both hash time and group (map) time** shows that, in most cases, the longer map time of parallel versions is not significant compared to the time savings from parallel hashing.



- Through Step 1 (hashing) and Step 2 (mapping) with locked map, all **parallel versions take less time** than sequential, with one exception. The N = 100,000 goroutines version in fine.txt has overhead that doubles the sequential execution time: 55 ms → 112 ms.

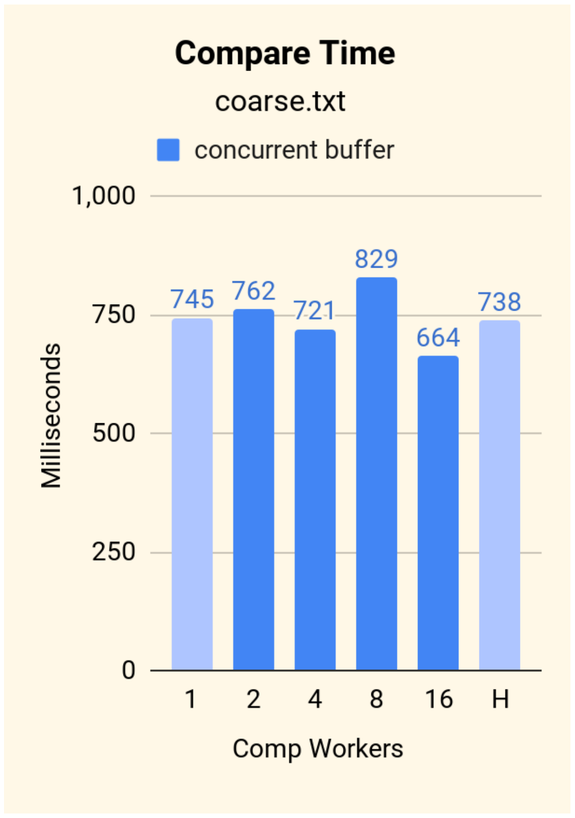
Step 3. Compare Trees

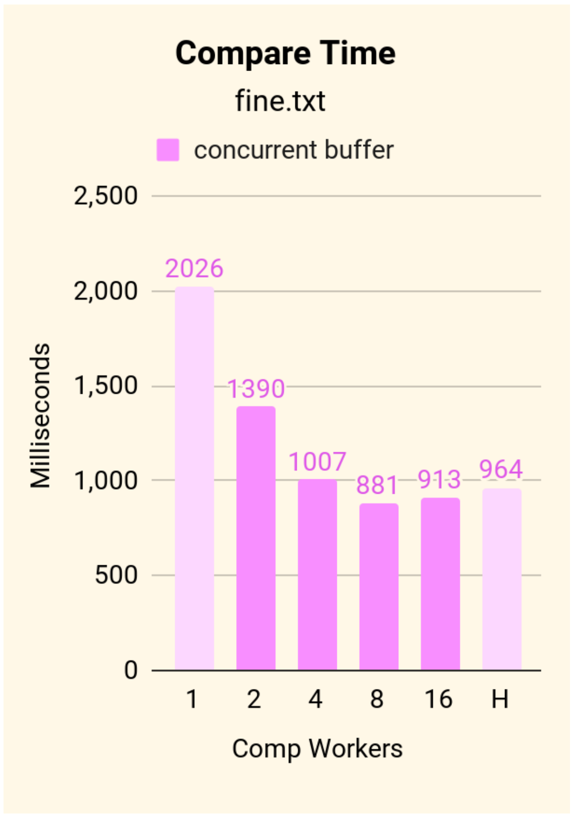
Lastly, I compared trees with the same hash to form *unique groups* of equivalent trees. As an **implementation choice**, I chose not to use an adjacency matrix. Instead, I made a `Group struct` to hold a slice of BST IDs, and `safeGroupList` with a lock for concurrent appends.

- **Goroutines with waitgroup** (first implementation)
  Launch H goroutines, where H is the number of hashes in the map. There is *one goroutine for each hash in the map*. It compares trees with IDs mapping to that hash and separates them into unique groups.
- **Custom Concurrent Buffer** (second implementation)
  Launch `comp-workers` goroutines that pull off work from a custom-made `concurrentBuffer` with fixed size equal to `comp-workers`. A main goroutine pushes hashes on the buffer while comp worker goroutines pop off hashes to process trees for.

Nicolas Ayllon
EID: nja842

The charts below show the results for parallelizing tree comparisons.





- In coarse.txt, parallelizing over hashes in the map shows **no improvement**. This is likely because all 100 trees collided into just H = 2 hashes, so work is split only in 2 parts.
- In fine.txt, parallelizing over hashes in the map **significantly decreases execution time** as the number of goroutines increases. Since the 100,000 trees covered all H = 1,000 possible values, there was a greater opportunity to divide work.

  The minimum execution time measured was for a concurrent buffer of size 8, but slightly increased for 16 and H = 1000, likely due to overhead without additional physical processors on the machine.

In the case of fine.txt, launching H goroutines captured most of the possible speedup. Although performance slightly improved by tuning the number of goroutines to 8, it’s not clear whether it justifies the additional complexity of creating `concurrentBuffer`.