CS380P: Parallel Systems

Lab #3

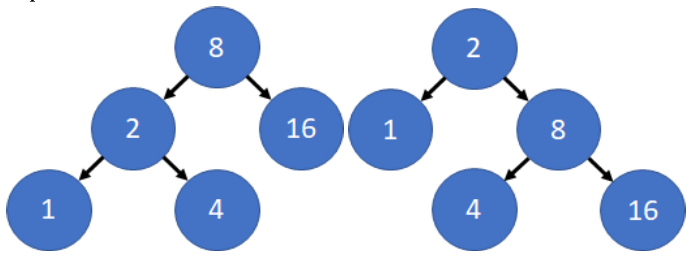
The goal of this assignment is to write a program in a language that restricts the programming model to simplify dealing with concurrency: Google's Go. You will use channels, go-routines, and signaling to compute binary search tree equivalence.

It is **highly** recommended that you read **all** instructions for this lab before you start coding.

Binary Search Trees

A binary search tree (BST) is a simple data structure where each node contains a data value and pointers to left and right subtrees. Each subtree is constrained such that the left substree contains only values smaller than its parent and the right subtree contains only values larger than its parent. While this constraint generally helps the average search time, it also limits where new values can be inserted (without displacing old values). This means the organization of the tree is dependent on the order in which values are inserted; two "identical" trees can have have very different layouts despite containing the same values.

Consider the two example trees below, each of which encodes the sort order 1, 2, 4, 8, 16. The tree on left, is produced by **insertion** orders in which 8 is inserted first, while the one on the right is produced by insertion orders in which 2 is inserted first. However, despite the fact that the data structures have different node-level organizations, an in-order traversal of the trees produces the same sort order, and the BSTs are therefore said to be "equivalent."



Algorithm

Given a set of binary search trees, we would like to determine which trees contain the same values. We can do this naively by comparing each tree to every other tree to see if they are the same (if traversing them in order results in the same values). However, this is highly inefficient.

Instead, we will first compute a hash of each tree by traversing it in order. We can then compare the hashes of the trees rather than the trees themselves. If the hashes of two trees are different, no further work is necessary. If the hashes of the trees are the same, the trees must be compared to see if they are really the same.

Deliverables will be detailed below, but the focus is on a writeup that provides performance measurements as graphs, and answers (perhaps speculatively) a number of questions. Spending some time setting yourself up to quickly and easily collect and visualize performance data is a worthwhile time investment since it will come up over and over in this lab and for the rest of the course.

Input files

We have prepared a few input files to help you with this assignment: simple.txt, coarse.txt, and fine.txt. Simple.txt is intended to be small to simplify debugging, especially via print statements. Coarse.txt contains a small number of very large trees, which should allow you to observe the benefits of parallelism without high amounts of overhead. Fine.txt contains many very small trees and should start to test the overhead and scalability of your implementation.

Step 1: Create a sequential solution

In step 1 of the lab, you will write a program that accepts command-line parameters to specify the following:

- `-hash-workers=` integer-valued number of threads
- `-data-workers=` integer-valued number of threads
- `-comp-workers=` integer-valued number of threads
- `-input=` string-valued path to an input file.

While the options allow you to specify a number of workers, your initial solution should just implement a single-threaded version of the algorithm. This greatly simplifies debugging and gives you a baseline against which to measure subsequent parallel versions. At this point, your implementation should:

- contruct BSTs as specified by the file
    - each line represents a different BST and ends in a newline character
    - the numbers on each line, which are separated by spaces, are the values contained in the BST
    - numbers should be inserted into the BST in the order provided

- generate a hash of each BST
- store the hashes and identify potential duplicates
- compare trees with identical hashes to determine their equality
- for each tree, store the IDs of each identical tree
  - a BST's ID is just the index of it in the input file

Step 2: Parallelize hash operations

There are three main steps that you should parallelize, the first of which is computing the BST hashes. During your first implementation, this should be done by spawning a goroutine for each BST. Once you have done this, try spawning `hash-workers` goroutines (threads) and having them iterate over the available BSTs. Try your code with several different values of `hash-workers` on both coarse.txt and fine.txt. Which implementation is faster (and by how much)? Can Go manage goroutines well enough that you don't have to worry about how many threads to spawn anymore?

For simplicity, we recommend you use the faster of your implementations before proceeding. With your hashing parallelized, you will now need to place these hashes in a shared location and identify duplicates. We believe the easiest way to do both of these is to use a map from hashes to BST IDs with that hash. However, since appending a new BST ID is not atomic, we must protect the shared data structure. For your first implementation, spawn a single thread to insert data into the map and have your hashing threads communicate with it via a channel by sending (hash, BST ID) pairs through it. Since channels are thread safe and only one thread is modifying the data structure at a time, this approach is entirely safe. Once you have done this, make an alternate implementation where the data structure is protected by a single lock and each thread must acquire the lock to add their result to the data structure (i.e. they now fight for the right to do the work of the helper thread before). Compare the speed of both implementations. Which approach has more overhead? How much faster are they compared to a single thread? Which approach do you find simpler?

(Optional) For some extra credit, make two more implementations which use fine grain synchonization to allow up to `data-workers` threads to access the data structure at once. Since multiple threads will be accessing the data structure, you will have to make sure that only one thread is modifying the entry for a particular hash at a time. Evaluate your performance on both coarse.txt and fine.txt. Was access to the shared data structure a bottleneck before? How do channels and locks compare now that there is more parallelism?

Step 3: Parallelize tree comparisons

For simplicity, we recommend you use the faster of your implementations before proceeding. With your hashes parallelized, it is now time to parallelize the final tree comparisons. To keep things simple, you can store BST equivalence using a 2D adjacency matrix where `array[i][j] == true` implies the BST with ID `i` is equivalent to the BST with ID `j`. After you have populated your map from hashes to BST IDs and initialized the adjacency matrix to `false`, have a single thread traverse it. If a hash is associated with only one BST ID, `k`, then you can simply assign `true` to `array[k][k]` and proceed. However, if a hash has multiple BST IDs associated with it, you will have to compare all of their trees for similarity. For your first implementation, just spawn a goroutine to do each comparison and write the result to the adjacency matrix if a match is found.

For your second implementation, you will spawn `comp-workers` threads to do the comparisons and use a concurrent buffer to communicate with them (work can be represented with a (BST ID, BST ID) pair of trees to compare). Since this is not a data structures course, it doesn't matter how the buffer is implemented as long as you make sure that it only holds up to `comp-workers` items at a time and isn't slow. Your buffer should contain a mutex to prevent concurrency errors when multiple threads try to access the buffer at once. The buffer should also contain two conditions to handle the cases where the main thread tries to insert work when the buffer is full and when the worker threads try to remove work when the buffer is empty. The threads should not "spin" and repeatedly check if the buffer is no longer empty or full. Evaluate your performance on coarse.txt (you can also test your code on fine.txt, but the large number of trees will result in an extremely large adjacency matrix, which will take significantly longer to access). How do the performance and complexity of both approaches compare with each other? How do they scale compared to a single thread? Is the additional complexity of managing a thread pool worthwhile?

Hints

Go includes a lot of useful packages. Argument parsing is trivial with the flag package. High-precision timing in Go can also be quite easy using the built-in time package. The io/ioutil package makes reading in file data easy. Once you strip the EOF character, you can parse the file with the strings and strconv packages. The sync package includes a lot of handy tools, including mutexes and conditions. It also includes the WaitGroup, which greatly simplifies waiting on a group of goroutines to finish. Finally, Go contains many familiar data structures, like arrays and maps. However, you may also want to check out the slice. It combines the power of lists, arrays, and slicing into a single data structure.

Deliverables

Please follow these supplemental instructions and submission guidelines.