CS380P: Parallel Systems

Lab #3: BST Comparison

Submission Guide

Read this page first for detailed instructions about how to do this lab.

When computing the hash of a tree, we will use the following function:

```
// initial hash value
hash = 1;
for each value in tree.in_order_traversal() {
  new_value = value + 2;
  hash = (hash * new_value + new_value) % 1000
}
```

The given input data (simple.txt, coarse.txt, and fine.txt) is used for exploring the difference between implementations to prepare the writeup. Hidden input data will be used by the autograder to test performance and correctness of each implementation. Your program will be tested for each step of the algorithm. Flags are used to select different implementations by the autograder. Make sure your program accepts the exact flag(s) for a particular implementation as detailed below.

You should submit your code and report packaged into a tar file on Canvas. If you do not complete the optional implementation in step 2.2, name your main go program `BST.go`. If you do complete it, name it `BST_opt.go` instead.

1. Computation of Hash of Each Tree

1.1 Flags

The following flag controls the number of goroutines to compute the hash of each BST

- `-hash-workers=<number of hash workers to hash BSTs>`: spawn `hash-workers` goroutines to iterate over the available BSTs to compute the hash.

When `-hash-workers` is set to 1, you do not have to spawn any goroutines. You can simply compute the hash of all trees in the main thread. When `-hash-workers` is the only flag provided, your program should only compute the hash of each BST without performing the other two steps (identification of hash duplicates and tree comparison). This is for the autograder to collect performance for this step only.

1.2 Output

For any number of hash workers, your program should output the time it takes to compute the hashes of all BSTs. Make sure that the time is measured after all goroutines are synchronized. The output should have the following form

```
hashTime: xxx
```

Note that the unit of the output time is second. Do not include tree construction time.

1.3 Testing

Your program will be tested for the following numbers of hash workers

```
(1,2,4,8,16,N)
```

where `N` is the total number of BSTs.

2. Computation of Hash Groups

2.1 Flags

The following flag controls the synchronization mechanism used by `hash-workers` goroutines to update the map

- `-data-workers=<number of workers to update the map>`: there will be `data-workers` goroutines trying to update the map.

Your program should select the proper implementation according to the following combinations of flags:

1. `-hash-workers=1 -data-workers=1`: This is the sequential implementation. Your program can compute the hash of all BSTs and update the map in the main thread.
2. `-hash-workers=i -data-workers=1(i>1)`: This implementation spawns `i` goroutines to compute the hashes of the input BSTs. Each goroutine sends its (hash, BST ID) pair(s) to a central manager goroutine using a channel. The central manager updates the map.
3. `-hash-workers=i -data-workers=i(i>1)`: This implementation spawns `i` goroutines to compute the hashes of the input BSTs. Each goroutine updates the map individually after acquiring the mutex.

The following combination of flags are for the optional implementation:

- `-hash-workers=i -data-workers=j(i>j>1)`: This implementation spawns `i` goroutines to compute the hashes of the input BSTs. Then `j` goroutines are spawned to update the map.
  - It is up to you how the hash workers are communicating with the `j` data workers. You can use a semaphore so that `j` of the `i` hash workers can update the map. Or you can use `j` central managers to collect (hash, BST ID) pairs from each hash worker via `j` channels.
  - When `j` goroutines try to update the map, make sure each hash entry is not updated by more than two threads at the same time.

2.2 Output

For each implementation, your program should output the elapsed time as well as the hash groups in the following form:

```
hashGroupTime: xxxxxx
hash0: id00 id01 ..
hash1: id10 id11 ..
..
```

`hashi` is the ith hash value and `idi0`, `idi1`,... are tree ids that have the same hash value `hashi`. Ids are separated by spaces. Do not print hash groups that have only one tree. Note that *hashGroupTime* includes the time for hash computations.

Do not show any output from the tree comparison if `-comp-workers` is not specified.

2.3 Correctness

The autograder will check the output hash groups to verify that both the hash values and corresponding tree ids match the reference output. Therefore, make sure you are using the hash algorithm from the top of this page to compute the hash of each tree's in-order traversal.

2.4 Performance

The autograder will run each implementation 10 times to obtain an average elapsed time to measure performance. Possible `i` values include 2, 4, 8, 16, N, where N is the number of BSTs in the input file. For the optional implementation, possible `j` values include 2, 4, 8, 16.

3. Tree Comparison

3.1 Flags

The following flag controls the number of goroutines doing the tree comparisons.

- `-comp-workers=<number of workers to do the comparisons>`: there will be `comp-workers` goroutines spawned to compare pairs of trees.

`-comp-workers=1` corresponds to the sequential implementation, in which your program can simply do tree comparisons in the main thread.

3.2 Output

Your program should measure how much time it takes to compare trees (*compareTreeTime*) and output the tree groups in the following form:

```
compareTreeTime: yyyyyy
group 0: id00 id01 ...
group 1: id10 id11 ...
..
```

Do not print groups that have only one tree.

It is ok that your program also outputs the hash groups from the previous step.

3.3 Correctness

The autograder will compare the tree groups with the reference output. Group ids do not have to match.

3.4 Performance

The autograder will execute each implementation 10 times and compute an average elapsed time to measure the performance of tree comparison. Possible `comp-workers` values include 1,2,4,8,16.