cs378: Concurrency: Lab 3 Writeup Template

You
Department of Computer Science
UT Austin

January 19, 2018

1 Step 1: Sequential Solution

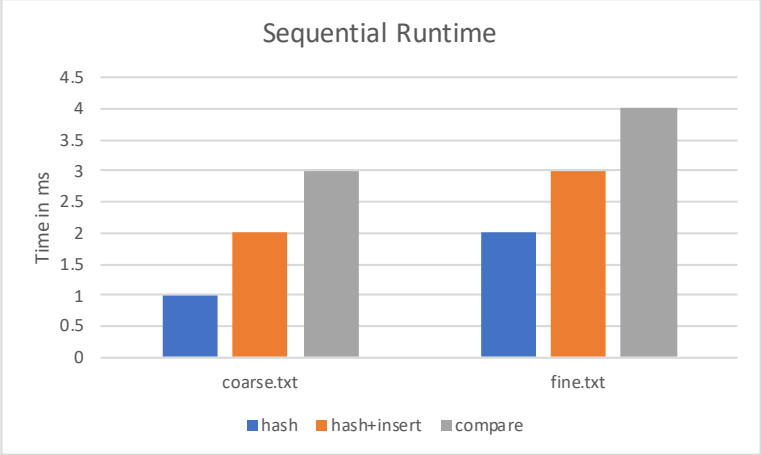

Figure 1: Step 1 sequential runtime graph. Note: data is not accurate.

Using Go's time package, report the time taken by your serial implementation to perform the following tasks. You can report the runtimes using graph(s) or table(s). See Figure 1 for reference.

- Generate the hashes
- Generate the hashes and insert them into the data structure
- Perform the tree comparisons and store the results in the adjacency matrix

2 Step 2: Parallelize Hash Operations

Using graphs, report the following data:

- The speedup over sequential from using goroutine and thread pool parallelism to perform hashing with both coarse.txt and fine.txt. Use several different values for the number of threads and make them interesting (powers of 2, the number of cores / hyperthreads on your CPU, and twice the number of cores / hyperthreads on your CPU are good). See Figure 2 for reference.

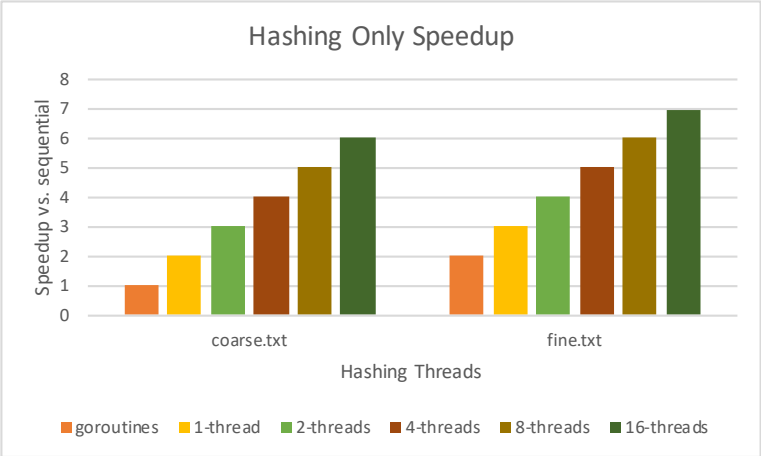

Figure 2: Step 2 hashing speedup graph. Note: data is not accurate.

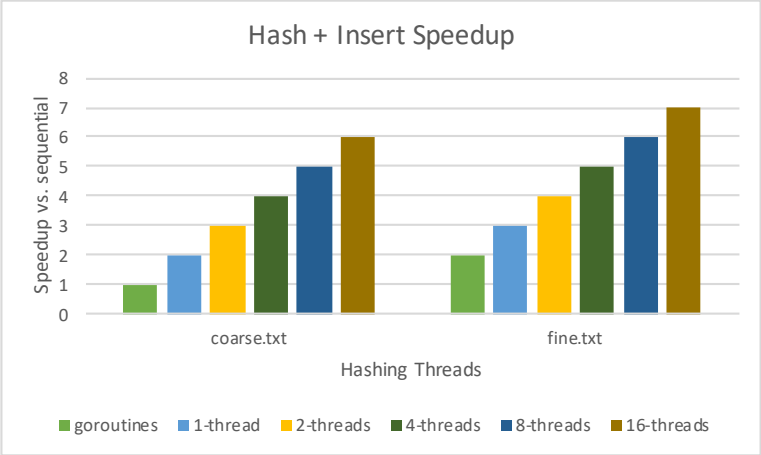

Figure 3: Step 2 hashing + insert speedup graph. Note: data is not accurate.

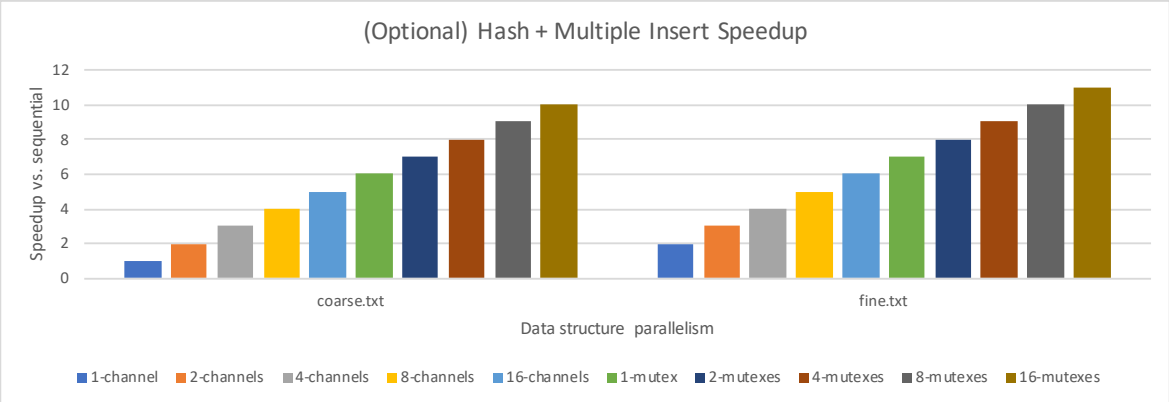

Figure 4: Step 2 (optional) hashing + multiple insert speedup graph. Note: data is not accurate.

- The same data as above, but including the time it takes to insert into the shared data structure. See Figure 3 for reference.
- If doing the optional fine-grain parallelism exercise, also report similar speedups over sequential for different numbers of channels and mutexes. See Figure 4 for reference.

In addition, answer the following questions with a brief (about 2 sentences) answer:

- Was either approach to parallelism significantly faster at hashing or hashing and inserting? If so, why do you think this was the case?
- Which approach had more overhead when accessing the data structure (if either)?
- How did the fine.txt input affect your scalability?
- If you implemented fine-grain data-structure access, did this significantly improve performance? If not, why do you think this was the case?
- After having tried both approaches, which do you prefer and why?

3 Step 3: Parallelize Tree Comparisons

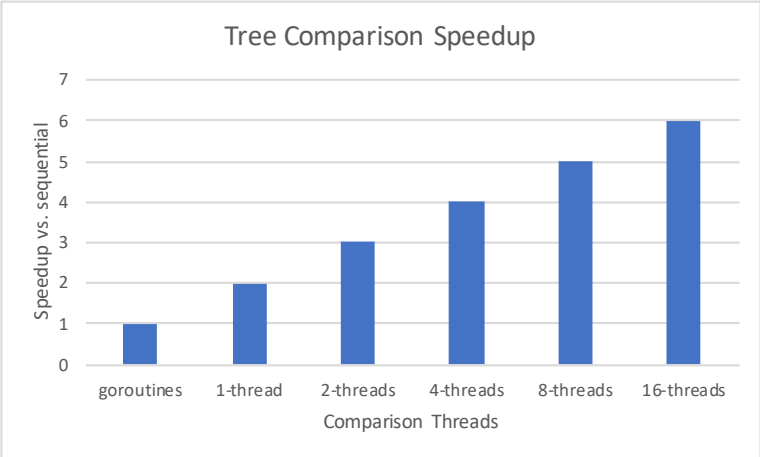

Figure 5: Step 3 tree comparison speedup graph. Note: data is not accurate.

Using a graph, report the speedup over sequential from using goroutine and thread pool parallelism to perform tree comparison with coarse.txt. Use several different values for the number of threads and make them interesting (powers of 2, the number of cores / hyperthreads on your CPU, and twice the number of cores / hyperthreads on your CPU are good). See Figure 5 for reference. In addition, briefly answer the following questions:

- Was either approach signficantly faster? If so, why do think this was the case?
- How did the complexity of the approaches compare? Did you find that one was much simpler?
- Now that your implementation is finished, how did you like Go? Would you still prefer working with C++ in the future (for performance and/or usability)?