cs378: Concurrency: K-Means Lab 2 Writeup Template

You
Department of Computer Science
UT Austin

September 14, 2018

1 Step 1: Serial Implementation

No deliverables in writeup.

2 Step 2: Coarse Parallelization

Using the random-n65536-d32-c16 sample input, create a graph of scalability for `pthread_mutex_t` *and* `pthread_spinlock_t` *for your solution from 1 to twice the number of physical processors on your machine for your solution at this step. Please normalize your measurements with the single-threaded solution from Step 1. This means that unlike in the first lab, where we simply reported execution time, this graph is speedup over sequential.*

- Provide a graph of speedup similar to Figure 4.

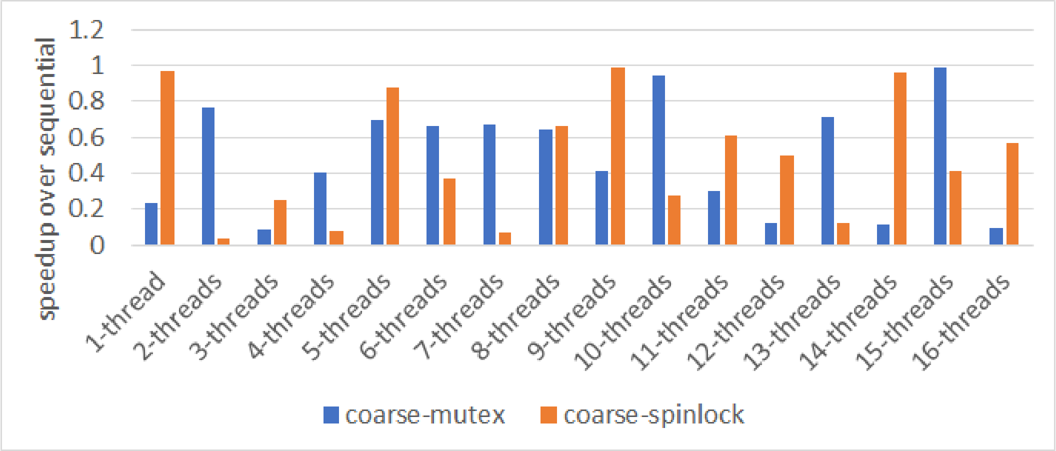

Figure 1: Step 2 scalability sample graph comparing coarse mutex against coarse spinlock. *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

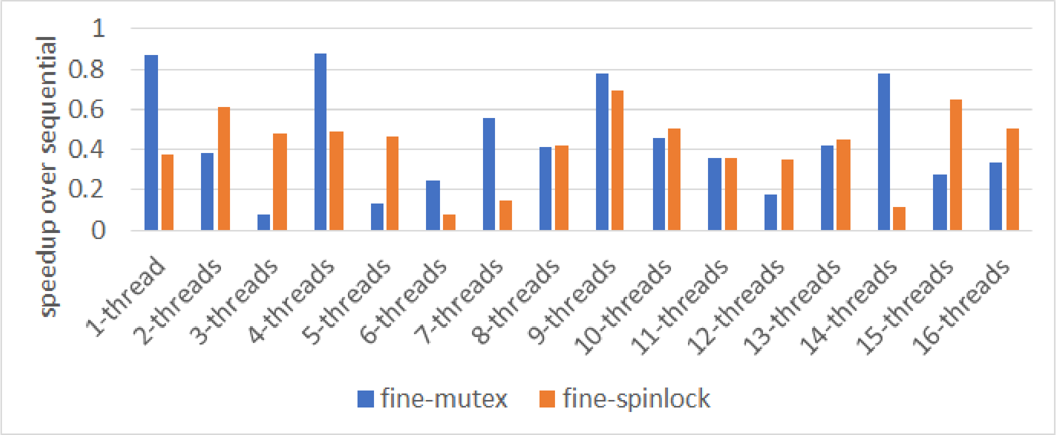

Figure 2: Step 2 scalability sample graph comparing coarse mutex against coarse spinlock. *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

3 Step 3: Finer-grain Synchronization

Again using the random-n65536-d32-c16.txt sample input, 20 maximum iterations, and a threshold of 0.0000001, create another speedup graph like the one from Step 2, where `pthread_mutex_t` *and* `pthread_spinlock_t` *primitives are used to synchronize en masse updates accumulated privately by each thread at the end of each iteration. Again your graph should be a speedup graph, where data are normalized to the single-threaded Step 1 implementation.*

- Provide a graph of speedup similar to Figure 5.

cs378: Concurrency: K-Means Lab 2 Writeup Template You
Department of Computer Science
UT Austin September 14, 2018

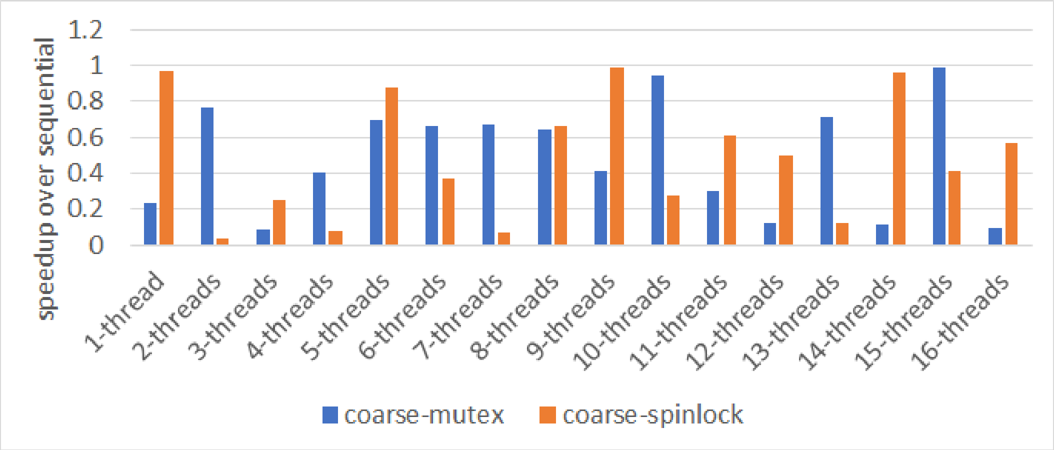

Figure 3: Step 2 scalability sample graph comparing coarse mutex against coarse spinlock. *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

4 Step 2: Coarse Parallelization

Using the random-n65536-d32-c16 sample input, create a graph of scalability for `pthread_mutex_t` *and* `pthread_spinlock_t` *for your solution from 1 to twice the number of physical processors on your machine for your solution at this step. Please normalize your measurements with the single-threaded solution from Step 1. This means that unlike in the first lab, where we simply reported execution time, this graph is speedup over sequential.*

- Provide a graph of speedup similar to Figure 4.

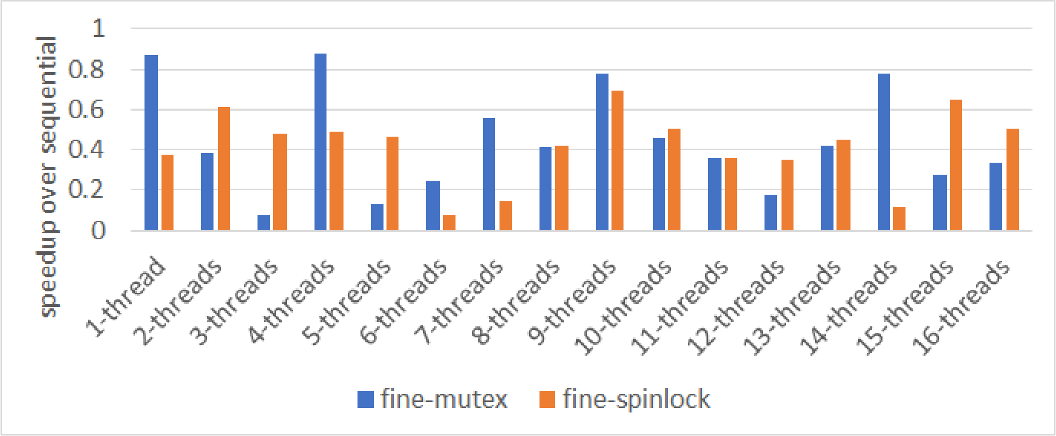

Figure 4: Step 2 scalability sample graph comparing coarse mutex against coarse spinlock. *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

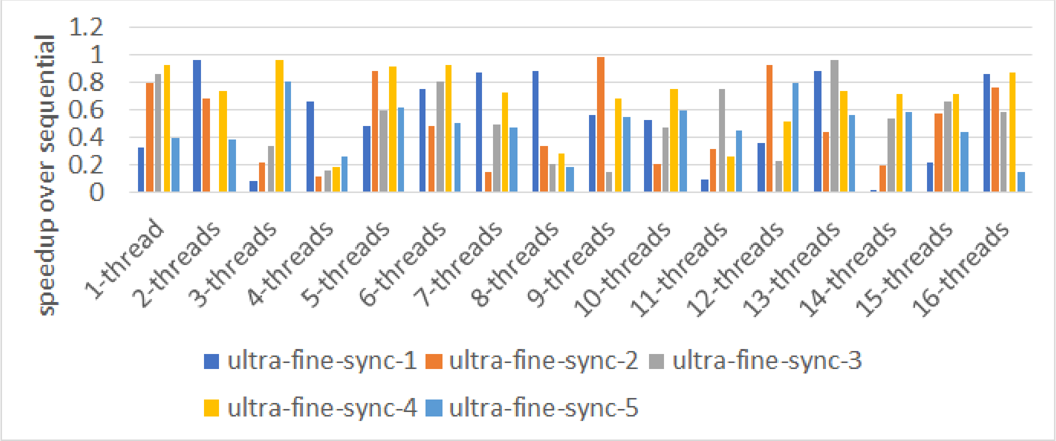

Figure 5: Step 4 speedup sample graph comparing your barrier to pthreads *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

5 Step 3: Finer-grain Synchronization

Again using the random-n65536-d32-c16.txt sample input, 20 maximum iterations, and a threshold of 0.0000001, create another speedup graph like the one from Step 2, where `pthread_mutex_t` *and* `pthread_spinlock_t` *primitives are used to synchronize en masse updates accumulated privately by each thread at the end of each iteration. Again your graph should be a speedup graph, where data are normalized to the single-threaded Step 1 implementation.*

- Provide a graph of speedup similar to Figure 5.

6 Step 4: Barrier Implementation

In your writeup, describe your design. Why did you choose the approach you chose? Under what conditions do you expect your barrier to perform best and worst. Finally compare the performance of your implementation using your barrier against your implementation using a pthreads barrier. This

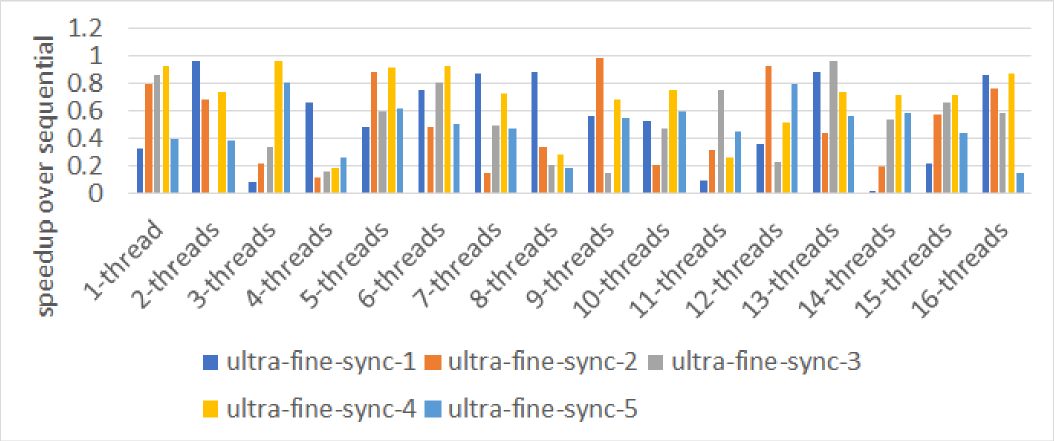

Figure 6: Step 5 speedup sample graph comparing who knows what synchronization techniques. *NOTE: the data in these graphs are random. If your data show different trends that's GOOD.*

comparision should involve generating the another version of the speedup graphs described in previous sections. What performance trends do you observe with your barrier? Please explain or speculate about why your barrier exhibits the scalability trends you observe.

- Provide a graph of speedup similar to Figure 6.

7 Step 5: Even finer-grain synchronization (optional)

In this optional step, you may, explore other ways to make synchronization even finer grained to reduce contention and increase scalability. Can you use CAS-based updates? HTM? Does padding data structures to avoid false sharing have any impact on your performance? Locks per-dimension? Functional decomposition instead of domain decomposition? Credit will be given for any reasonable solution that undertakes this section, as long as the solution is still correct: trying to improve scalability is a worthwhile endeavor, even if you don't succeed. Include graphs, along with some conjecture explaining the performance behavior of your implementation(s).

- Provide a graph of speedup similar to Figure 7.