CS 446.1001 Assignment 2 Questions

Robb Northrup

October 3, 2024

1 Procedures

1. Run with different values (1000 1,000,000)
2. Run with different number of requested threads

2 Questions

1. **What were your observations?**

Number of Threads	oneThousand.txt time elapsed (ms)	oneMillion.txt time elapsed (ms)
1	0	2
2	0	2
4	3	2
8	1	1
16	3	2
32	2	8
64	11	5
128	11	19
256	23	22

Table 1: Number of threads vs. time elapsed

Based on the data from table 1 (from my observations of the final runtime given the different parameters), it seems that a larger data set means an overall runtime increase. So far as thread count is concerned, a higher number of threads reduces the time it takes to complete the task, up to a certain point were it then inhibits the program. 8 Threads seems to be the sweet-spot.

2. **Are these observed behaviors reasonable?** These behaviors are entirely reasonable: increasing the amount of numbers to be processed will increase the number of operations necessary to sum the dataset. Using more threads generally improves performance as each thread works on a portion of the data, concurrently. That being said, at a certain point, a high thread count degrades performance. This is likely due to thread management overhead (thread creation + synchronization), thus increasing runtime.

3. **What considerations did you make for your implementation of the critical section? Provide your reasoning.** This section was concerned with updating the shared totalSum variable (accessed by all threads). My considerations were the following: thread safety, mutex lock, and minimizing lock duration. For safety, because multiple threads are working at the same time, I had to avoid race conditions. Synchronization was necessary. I couldn't let two threads (or more) write to the sum at once. As far as the mutex lock is concerned,

```
pthread_mutex_lock
pthread_mutex_unlock
```

were used. This helps make sure that only one thread updates at a time. Lock duration was minimized by placing mutex lock around minimal code.

4. **What would happen if instead of having the threads loop over the int array to compute a local arraySum, and finally update the totalSum, we were instead just directly adding to totalSum each time we were accessing an array element within each Thread (i.e. within the Thread's for loop)? Why?** A few of issues would arise. Poor performance would be a primary concern, as this would be a product of frequent locking and unlocking in the program. This would be a serious bottleneck on runtime efficiency, expecially with a larger number of threads. Threads would be blocked frequently, waiting on others to unlock. Overhead would also be increased drastically. Multiple threads would likely fight over the lock, which means that threads would be for waiting for critical section access.