CMSC 23010: Homework 3 Final Writeup and Theory

Will Thomas

November 11, 2022

Introduction

The goal of the following documentation is to discuss design changes, implementation challenges, correctness outcomes, and performance outcomes for a program shows the differences in performance between certain locks.

Design

Lock Design

Each lock implemented in the code had its own struct (except for PTHREAD_MUTEX_t, as it is builtin). The TAS and TTAS lock structs each consisted of a single volatile int called state, while the ALock struct has a volatile int tail, an int size, and a volatile int array called flag. Beyond that, for ease of coding, there is a union type lock_t containing a field for a tas lock, a ttas lock, a pthread mutex, and the Anderson lock.

Each lock also has its own associated functions. TAS and TTAS each have a lock and unlock function which follows closely to the scheme laid out in the textbook. The Anderson Lock has a lock and unlock function as well as an initialize function. The initialization function is to set the proper values for the queue size as well as to allocate space for the queue itself.

Every function written to implement these locks takes advantage of gcc sync builtins, most notably __sync_synchronize(), __sync_lock_test_and_set(), and __sync_fetch_and_add().

Serial Functions

serial_count

serial_count is extremely simple; it declares a volatile int count, sets it to 0, and then increments it in a while loop until it reaches the target number.

Concurrent Functions

concurrent_count

concurrent_count essentially does what serial_count does except that it handles all of the needed conditionals on which lock functions to call. The threads that are passed to it have as arguments the number to count to, the thread id, a code referring to which lock is to be used, a pointer to the lock itself, an integer that is a counter for number of times in the critical section, and a pointer to the larger shared volatile int count. From there, the thread will call the designated lock, wait until it is given control of the lock, increment the counter, and then releasing the lock.

Main

The program itself takes the following arguments:

1. big_number, the number to count to.
2. version, which can be S or C, S being serial and C being concurrent.
3. n, or numthreads.
4. lock_type, which can be 0, 1, 2, 3, which are tas, ttas, mutex, and alock, respectively.
5. test_version, which can be 0 or 1. The only difference is that with the value 1 the program will record each threads' times in the CS into an outfile.

The main function parses the inputs and runs the correct functions accordingly. If the inputs specify that the code is to be run serially, the program does not bother to unpack the values for n, lock_type, or test_version as they are irrelevant. In the concurrent conditional branch, the thread arguments are all set up, and the specified lock is created. A for loop is entered that launches off each thread, and then another for loop joins them. At the end of both the concurrent and serial branches the computation time is written into an outfile. If test_version is set to 1, in addition to the computation time each thread's number of times in the critical section is also written to an outfile.

Testing

Correctness testing is done by a python script called correct_test.py. It contains two functions. Both tests' working is dependent on certain print statements being uncommented within main. Basically, whenever the counter is updated, the new value should be printed out. I had to do this instead of writing to a file because for whatever reason fprintf was far too slow and would start to skip entries.

Regardless, the first function will run the program over the same inputs as required in the Lock Scaling performance test. That is, $n \in \{1, 2, 4, 8, 14\}$ and every type of lock. Each time it runs the program with a certain allocation of inputs, it checks the outfile against the serial outfile to make sure every number is present and in order.

The second function runs over the same range of inputs, except after each running of the program it will check two things:

1. The sum of every threads' count of times in the critical section adds up to the target number.
2. The last value in the outfile is the target number.

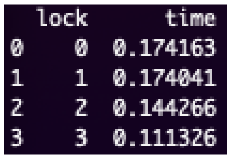
Beyond this, it also will run over all the values of n for the Anderson lock and print each thread's number of times in the critical section to show that they are in general very similar (although not always). This is kind of a weak way to demonstrate FIFO behavior, but the system itself is complicated enough that it would require a significant amount of work to prove more than this.

Performance

To test performance, I uploaded my Makefile, my main.c file, my correct_test.py python script, experiments.py which runs the required experiments, the needed utils in a function called utils, and three output directories needed for the correctness tests and experiments. The number counted to was chosen to be 1,000,000, and each datapoint is the median after the required inputs were run 6 times. The performance is as such:

Idle Lock Overhead

1. Table:



```
   lock      time
0     0  0.174163
1     1  0.174041
2     2  0.144266
3     3  0.111326
```

2. Analysis

   (a) TASLock

   In the design document, I predicted that TASLock would have the smallest overhead. This is clearly the case from the table, although I was incorrect in predicting that it would only take marginally longer than the serial code. As it turns out, it took significantly longer than the serial code. I believe that the reason for this is overhead involving in invoking atomic operations. Without going into heavy specifics with how the sync operations work, one would assume that there is a time intensive check to make sure that memory operations are ordering correctly. Beyond this, it also disables any optimization effects that the compiler is free to enact on the serial code, which I would presume to be quite significant.

   (b) TTASLock

   I predicted that TTASLock would perform almost identically to the TASLock. I was correct in this capacity, although once again I failed to realize the overhead involved in atomic operations as well as the detriment of disabling optimizations.

   (c) Mutex

   Due to not knowing significant details on how pthread_mutex_t works, I predicted that it would be "correct, portable, and quite fast." As seen in the table, it is roughly 82% as fast as the TAS and TTAS locks, which makes sense, especially considering that there is probably considerably more overhead going on under the hood. Despite all of this, performance slowdowns stemming from ordering and lack of optimization make themselves apparent.

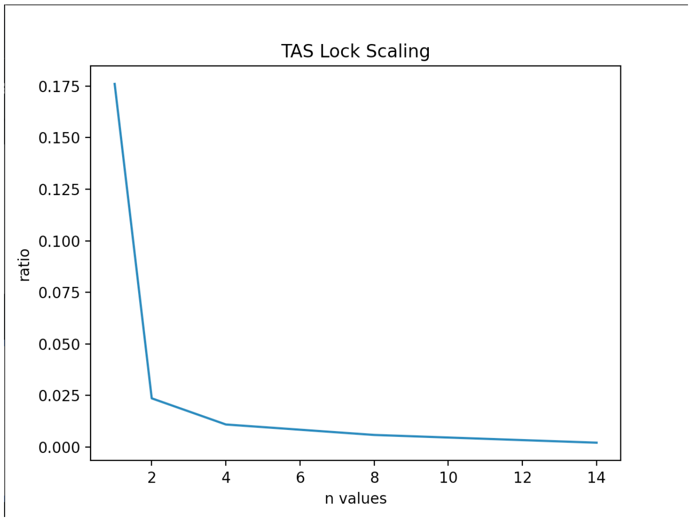
   (d) ALock

   I hypothesized ALock to be the slowest due to its substantial overhead (mainly in creating the array, setting all values to what they need to be). This turned out to be correct.

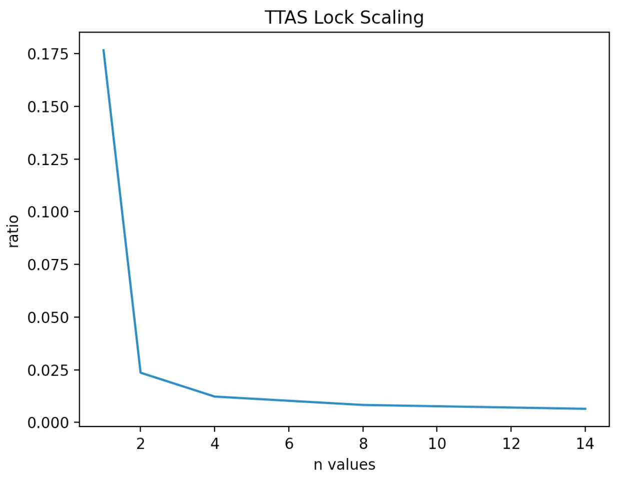
Lock Scaling

1. Plots

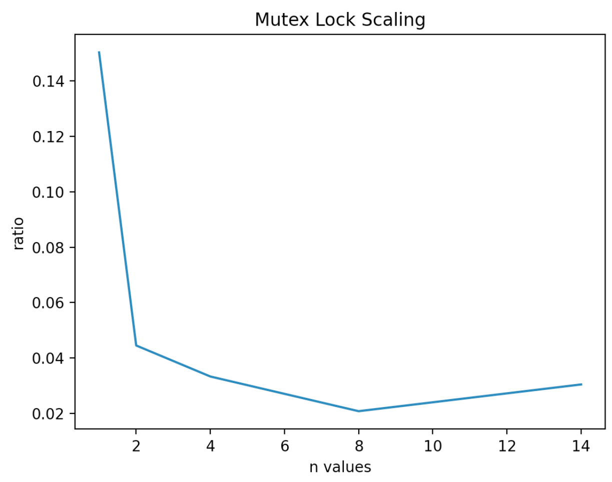
   TAS Lock Scaling:



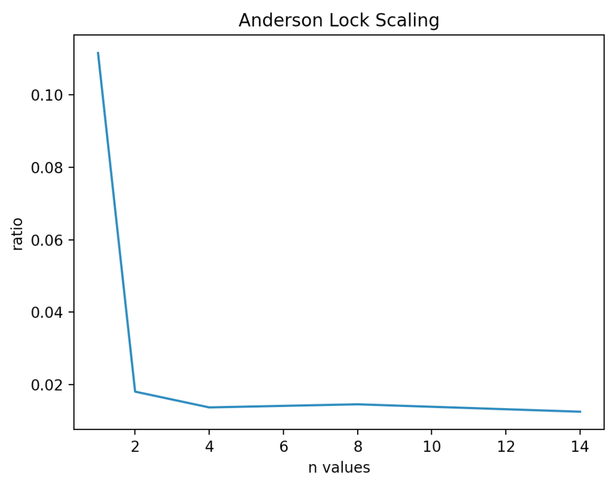
TTAS Lock Scaling:



Mutex Scaling:



ALock Scaling:



2. Analysis

As we have seen, TASLock is very much prone to horrible performance. This is an issue due to processor's cache busing system and the need to update values after every test and set. With two threads, it is already plummeted to 2.5% as fast as the serial computation, and it continues to decline as threads increase.

TTASLock performs much the same, which implies that the benefits from not having the cache update issues are very much obviated by the issues rising from the collective cache miss that occurs after a thread leaves the critical section.

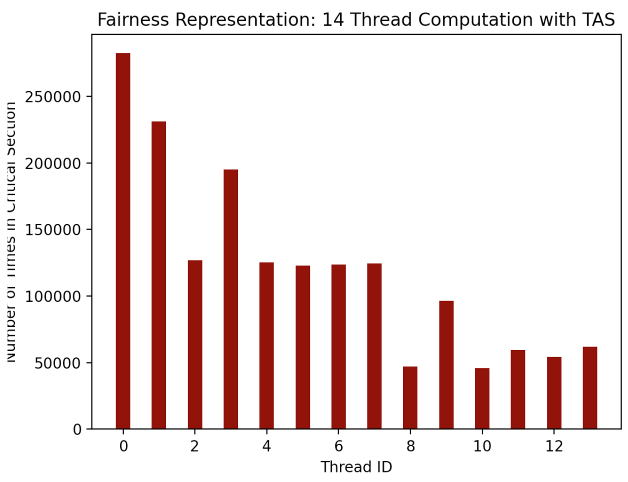
As predicted, the Mutex lock performs by far the best; the initial plummet after adding a second thread is the most shallow out of all of the locks, putting it at a relatively high performing 4% as fast as the serial code. What is perplexing, however, is the strange improvement in speed after adding 8 threads. The speedup continues to the point that having 14 threads is almost as fast as having 2 threads. Once again, it is hard to say with any certainty what is causing this without understanding the inner-workings of the Mutex lock.

The Anderson Lock has the worst initial plummet out of all of the locks, but it plateaus and holds far better than TASLock or TTASLock do, staying at around 2% as fast as serial. As such, I do not think it is entirely incorrect to say that it has the second best performance. I believe that the reason for this is that the queue system, especially with padding, requires a good amount of overhead due to the checking and memory ordering required, but then manages to stay effectively constant with increasing numbers of threads.

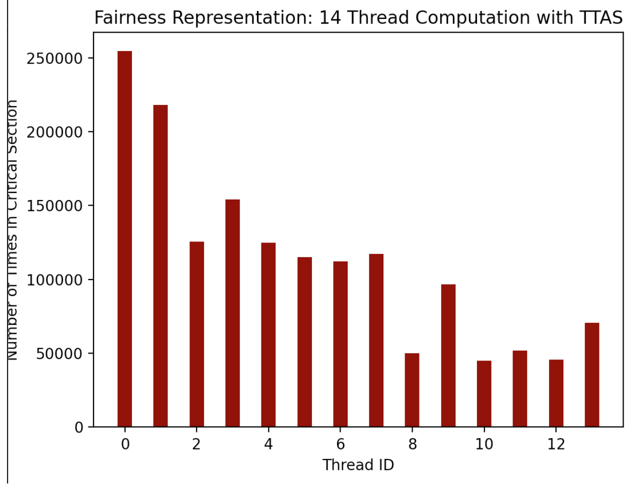
Fairness Testing

1. Plots

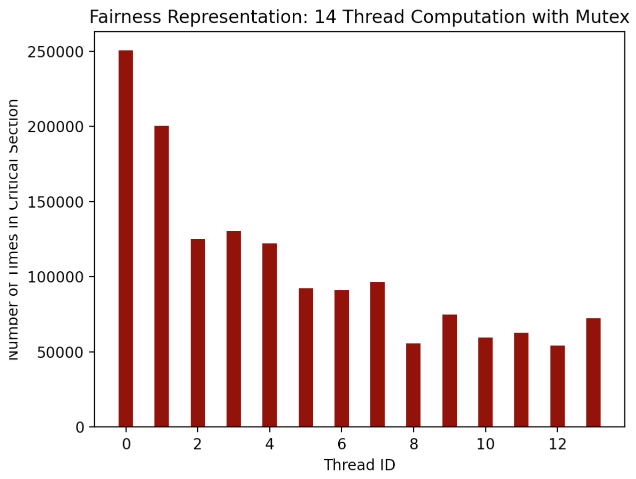
   TAS Fairness



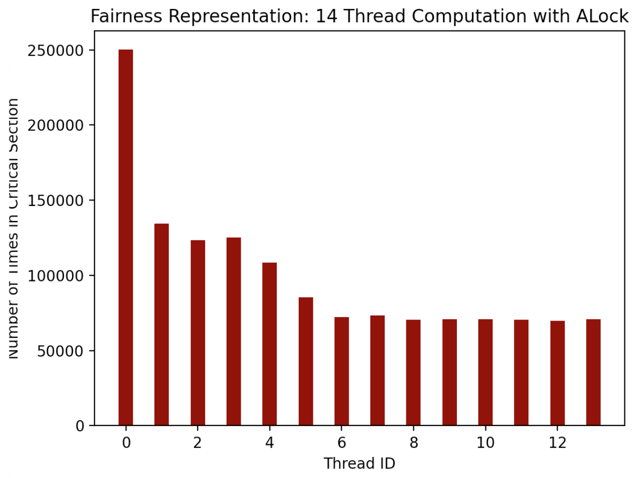
   TTAS Fairness



   Mutex Fairness



ALock Fairness



2. Analysis

As hypothesized, TASLock and TTASLock both seem to have the highest variance in number of times in the critical section per thread. This makes sense given the lack of any ordering guarantee; it's a nearly random race condition. This being said, the first thread in all cases (including Anderson lock) seems to have significantly larger amount of CS entries than the rest of the threads. This is most likely a direct effect of how threads are allocated and sent off. There is a for loop that creates the threads and sends them to start computing the concurrent function, meaning that there is a considerably amount of time passing where the contention is low before all threads are attempting to acquire the lock.

Interestingly, Mutex seems to perform similarly to TAS and TTAS. Once again, we do not know the details on how the mutex lock is implemented, meaning we cannot say for sure whether or not there are any FIFO guarantees.

Subject to the situational effect of the first thread entering the CS more often than the rest, it is clear that the Anderson Lock is the most fair, especially among the last 8 threads (they are nearly level in the histogram). Any assumptions I made previously about statistically insignificant differences in CS entry proved to be incorrect, although this makes sense given the occurrence of degenerate effects as well as the first few threads having more time to work with relatively low contention.