# Parallel Programing HW3a Writeup
Andrew McCarthy

Source files are located in the hw3a directory. My code is counter.c, taslock.c/.h, alock.c/.h, and clhlock.c/.h. Instructions on how to compile and run are located under the "Details" section

- Overview

This program has either one thread or multiple threads increment a counter until a specified value set by the user is reached. The serial version has a single thread increment the counter until it reaches the set value. The parallel version has multiple threads incrementing the counter concurrently. Each thread acquires a lock (the type of lock is set by the user) before incrementing the global counter. The program also has a timed counter version where the threads (or single thread if serial) increments the counter for a specified amount of time. This function is only used in testing. The program takes in four required parameters each indicated by a flag: -n followed by number of threads, -p followed by nothing (include this flag to enable parallel version, don't include it to enable serial version), -l followed by the BIG value, and -t followed by the lock type (0 for mutex, 1 for TAS, 2 for array lock, and 3 for clh lock). The parameters can be provided in any order. The program can also run performance tests by making the first argument "test" and the second an int representing which test to run (0 for lock overhead, 1 for parallel lock test, and 2 for timed counter test).

- Design

The TAS lock continually loops and checks the status of the lock variable until it is 0. Once it is 0, it sets the value to 1 and executes the critical section. Once it is done, it sets the variable back to 0. The lock function uses the gcc atomic builtin __sync_fetch_and_or to check the status of the lock and acquire it once it is free. The unlock function uses the gcc atomic builtin __sync_and_and_fetch to release the lock. My implementation is the same as the one I outlined in my design document.

The array lock uses an array called aLockFlag to indicate which thread currently holds the lock. In the lock function, the thread spins on its location in the array until the previous

thread sets its location to 1. The lock function also sets a currSlot pointer variable which represents which thread in the array currently holds the lock. The __sync_fetch_and_add gcc atomic builtin is used when setting the currSlot variable. In the unlock function, the currently held slot is set to 0 and the next element in the aLockFlag array is set to 1. This function is nearly identical to the implementation in my design document, except I used __sync_fetch_and_add instead of fetch_and_inc.

The clh lock uses qnode objects to represent if a thread has a acquired the lock or not. Each of these objects has a locked, id, and previousNode attribute. It also uses a clhLockTail variable to represent the tail of the qnode array. When a thread is waiting for a lock, it sets the locked field of its qnode to 1 and spins on its predecessors locked field. Before it does this, the thread sets its node as the tail of the queue and acquires a reference to its predecessors node. To release the lock, the thread sets its locked value to 0 and makes its predecessors node its node for future lock accesses. This implementation is also very similar to the one in my design document, except I use a different gcc atomic builtin.

For the counter portion of my code, the threads acquire the lock before checking if there are further increments to make. While this adds extra overhead, it ensures that the counter value will always be correct. The main function for the program processes the user parameters, then creates the threads to increment the counter. Once the counter has reached the specified value, all the threads are joined and the run time is reported.

I also included a tryLock function for each of the locks. The only difference between this and the normal lock function is that if the lock cannot be acquired, the function returns false instead of waiting for the lock to be released.

- Details

To create the executable (called “counter”) run make. To remove the executable and all the .o files run make clean. To run the program in normal mode, pass in the following arguments. -n (number of threads) -p (include this to run it in parallel mode. Not including it will run the program in serial mode) -l (BIG value in milliseconds) -t (lock type: 0 for mutex, 1

for TAS, 2 for array lock, and 3 for clh lock). These arguments can be provided in any order. Here is an example execution: ./counter -n 8 -p -l 100000 -t 1.

- Correctness Testing

For correctness testing, I printed out how many times each thread incremented, to what value counter was incremented to, and how long it took them to do so. I found that the increments were evenly distributed between the threads in all my custom locks at high BIG values (the difference was no larger than 200). With the mutex lock however, the difference was more severe (a difference of about 500-1,000). To achieve correctness, I put an if-statement checking to see if counter had reached its BIG value inside the lock. While this added extra overhead, it ensured exact correctness everytime. Putting this check outside of the lock resulted in a counter value that was always nthreads-1 larger than BIG. this is because a thread could be waiting for the lock to be released while the counter was successfully incremented to BIG. Then, it would acquire the lock and increment even though counter doesn't need anymore increments. I found that all my locks passed my correctness tests for multiple combinations of parameters

- Performance Testing

For my performance tests, I ran the two tests outlined in the project description as well as my own custom test. For my test, I had the threads count for a defined amount of time then I returned the counter value. I ran my test with 10 threads and for 1, 5, and 10 seconds. The results of my tests are listed below. Also, the ratio in my output is (limit/parallelTime)/(limit/serialTime).

- Lock Overhead Test

ratio for mutex lock with BIG set to 1000 is 0.013245
ratio for tas lock with BIG set to 1000 is 0.048387
ratio for array lock with BIG set to 1000 is 0.051724
ratio for clh lock with BIG set to 1000 is 0.101695
ratio for mutex lock with BIG set to 50000 is 0.153891

ratio for tas lock with BIG set to 50000 is 0.165212
ratio for array lock with BIG set to 50000 is 0.283120
ratio for clh lock with BIG set to 50000 is 0.278654
ratio for mutex lock with BIG set to 700000 is 0.158727
ratio for tas lock with BIG set to 700000 is 0.169726
ratio for array lock with BIG set to 700000 is 0.272330
ratio for clh lock with BIG set to 700000 is 0.290571
ratio for mutex lock with BIG set to 100000000 is 0.170666
ratio for tas lock with BIG set to 100000000 is 0.181940
ratio for array lock with BIG set to 100000000 is 0.245301
ratio for clh lock with BIG set to 100000000 is 0.310517
ratio for mutex lock with BIG set to 500000000 is 0.170693
ratio for tas lock with BIG set to 500000000 is 0.181905
ratio for array lock with BIG set to 500000000 is 0.236894
ratio for clh lock with BIG set to 500000000 is 0.310481

- Parallel vs serial test

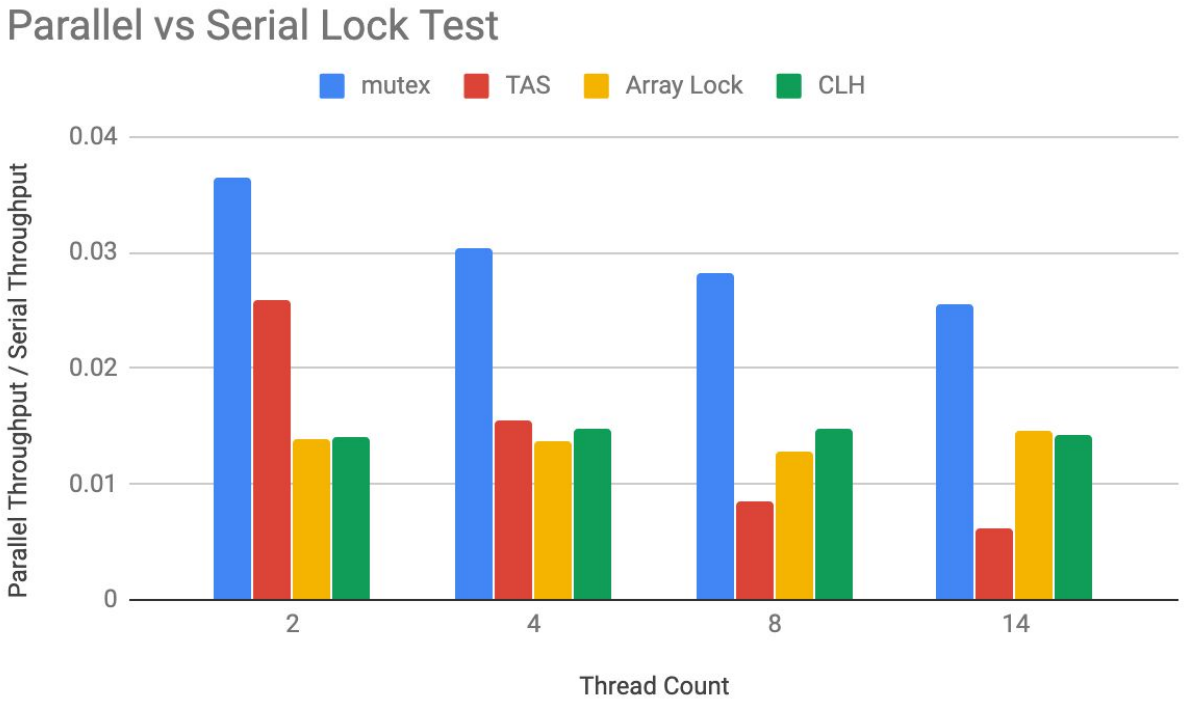
ratio for mutex lock with 1 threads is 0.143586
ratio for tas lock with 1 threads is 0.182306
ratio for array lock with 1 threads is 0.124575
ratio for clh lock with 1 threads is 0.310398
ratio for mutex lock with 2 threads is 0.036446
ratio for tas lock with 2 threads is 0.025887
ratio for array lock with 2 threads is 0.013841
ratio for clh lock with 2 threads is 0.013977
ratio for mutex lock with 4 threads is 0.030343
ratio for tas lock with 4 threads is 0.015504
ratio for array lock with 4 threads is 0.013688
ratio for clh lock with 4 threads is 0.014691
ratio for mutex lock with 8 threads is 0.028174
ratio for tas lock with 8 threads is 0.008445
ratio for array lock with 8 threads is 0.012800
ratio for clh lock with 8 threads is 0.014799
ratio for mutex lock with 14 threads is 0.025611
ratio for tas lock with 14 threads is 0.006241
ratio for array lock with 14 threads is 0.014615
ratio for clh lock with 14 threads is 0.014242

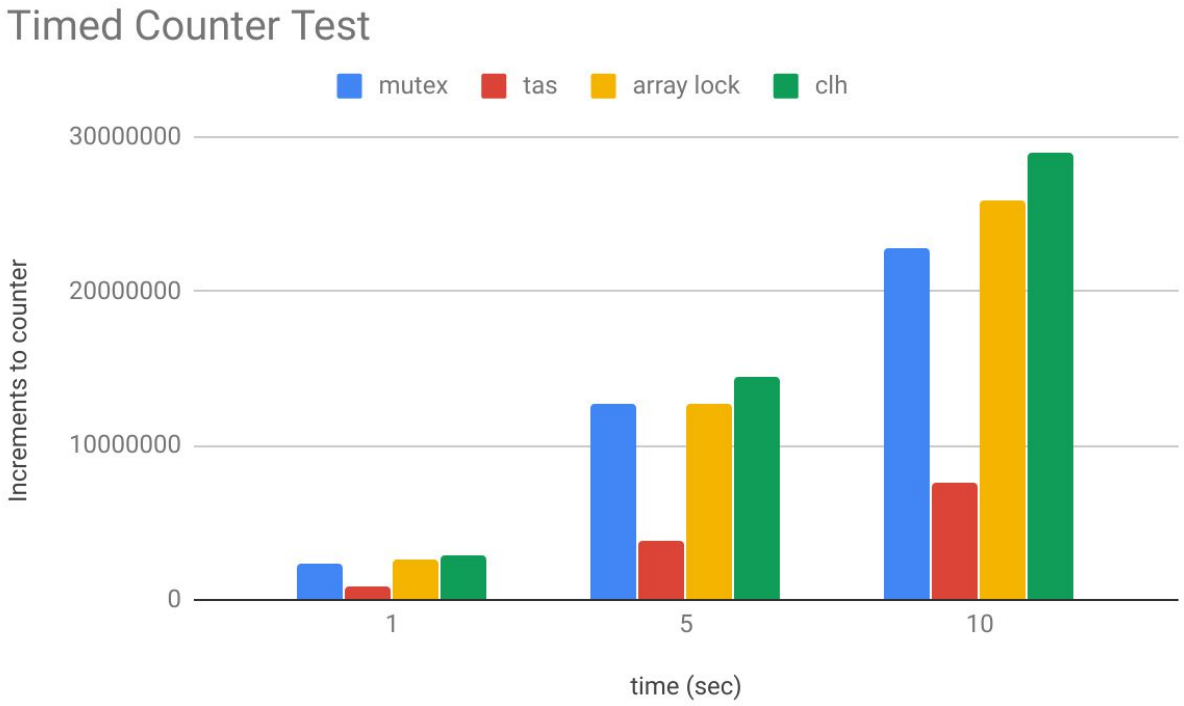
- Timed counter test

increments for serial with 1.000000 secs is 17913427.000000
increments for mutex lock with 10 threads with 1.000000 secs is 2332508.000000
increments for tas lock with 10 threads with 1.000000 secs is 862800.000000
increments for array lock with 10 threads with 1.000000 secs is 2610017.000000

increments for clh lock with 10 threads with 1.000000 secs is 2880621.000000
increments for serial with 5.000000 secs is 90691763.000000
increments for mutex lock with 10 threads with 5.000000 secs is 12644901.000000
increments for tas lock with 10 threads with 5.000000 secs is 3882430.000000
increments for array lock with 10 threads with 5.000000 secs is 12637832.000000
increments for clh lock with 10 threads with 5.000000 secs is 14468683.000000
increments for serial with 10.000000 secs is 181246625.000000
increments for mutex lock with 10 threads with 10.000000 secs is 22769891.000000
increments for tas lock with 10 threads with 10.000000 secs is 7601474.000000
increments for array lock with 10 threads with 10.000000 secs is 25882076.000000
increments for clh lock with 10 threads with 10.000000 secs is 28911215.000000

- Graphs
  Shown below are the graphs illustrating the results of my tests.


Parallel vs Serial Lock Test
mutex
TAS
Array Lock
CLH
0.04
0.03
0.02
0.01
0
Parallel Throughput / Serial Throughput
2
4
8
14
Thread Count


Timed Counter Test
mutex
tas
array lock
clh
30000000
20000000
10000000
0
Increments to counter
1
5
10
time (sec)

- Performance Testing Results

For the lock overhead test, my findings did not line up with my hypothesis. I thought that the simplest lock, TAS, would have the best performance with only one thread. However, my results show that CLH and array performed the best at large BIG values (around 20-30% better). I now see that this is because even though the TAS lock is simpler, there is more overhead when acquiring a lock even if it is uncontended. Interestingly, the mutex lock performed worse than every other lock. I believe this is due to compiler optimizations that don't occur if there is only one thread.

For the parallel vs serial test, my findings lined up with my hypothesis. I thought that the CLH and array locks would perform better at high thread counts as they have less contention than the TAS lock. At 8 and 14 threads, the array and CLH lock performed about 40-50% better than the TAS lock. At low thread counts however, the TAS lock performed the same or better than the array and CLH lock. This lines up with my hypothesis from the previous experiment. When contention isn't a big issue, the TAS lock will perform the best as it is the least complex. I tried to reconcile the difference in results between the second and first test, and ultimately decided that it may have something to do with how the compiler compiles single threaded counters. When there is only one thread, certain optimizations are performed that make the CLH and array lock faster. When there is more than one thread however, these optimizations don't occur and my hypothesis becomes valid. Also, the mutex lock performed significantly better than the CLH and array lock (about 50%). This also lines up with my hypothesis.

For the timed counter test, I wanted to see if my hypothesis from the previous test was still valid. My results validated my hypothesis that the CLH and array perform better than the TAS lock at high thread counts due to their low contention. The CLH and array lock consistently performed 120-150% better than the TAS lock. Interestingly, the mutex lock performed worse than the CLH and array lock (about 10-20%). I suspect that this may be due to compiler optimizations that don't occur if the threads aren't counting to a set value.

Ultimately, my tests validate most of my hypotheses while also revealing possible information about multithreaded compiler optimizations.