CMSC 23010 Homework 3a Programming Final

Ruolin Zheng

May 12, 2019

The svn submission in the `hw3a` directory includes:

- `Makefile` - instructions for the compiler
- `include/` - a folder containing header files for `driver, lock_utils, tas, ttas, alock, clh`
- `src/driver.c` - the main program which parses command line arguments and run either SERIAL or PARALLEL with the specified LOCK
- `src/lock_utils.c` - a wrapper struct that allows `tas, ttas, mutex` to use the same interface as in the function `worker_generic` defined in `driver.c`; it also allows `alock, clh` to use a minimally modified interface as in the functions `worker_alock, worker_clh`
- `tests/driver_output.c driver_output.h` - the same program as `src/driver.c` except that it writes the resultant Counter array to `serial_res.txt, tas_res.txt, ...` Please see **Counter Correctness Testing** below for details
- `tests/test_driver_out.py` - automated Python script for running multiple trials of `test/driver_output` and comparing the results among SERIAL, PARALLEL and SERIAL-QUEUE
- `tests/test_alock.c, test_clh.c` - very basic correctness tests for `alock, clh` using the `criterion` framework
- `result/` - data from running on SLURM stored in `.txt` and `.csv`
- `plot.ipynb` - Jupyter Notebook file for generating plots

1 Running the Program and the Tests

Please run `make clean` and `make` in the `hw3a` directory. This will create the following binary executables: `./driver, tests/driver_output, tests/test_locks`.

1.1 The Program: driver

Usage:
`./driver -b BIG -n NUM_THREADS -l [tas|ttas|mutex|alock|clh]`
To run SERIAL: `./driver -b BIG -n 0`

1.2 Lock Correctness Unit Testing: test_locks

Please run this binary located in `tests/` as `./test_lock --verbose`.
Because it is uncertain that how `criterion` handles multi-threading, these tests are only basic sanity checks that ensure that the locks are initialized correctly; one single thread can lock and unlock the locks multiple times; and that two threads spread out in time can still lock and unlock. **Concurrent correctness is tested below.**

1.3 Counter Correctness Testing: test_driver_output.py

Please make sure there is a binary named `driver_output` inside `tests` and then run `test_driver_output.py` inside `tests`.
`driver_output` accepts the same parameter as does `driver`, but writes the resultant **Counter array** for correctness testing. This script invokes `./driver_output -b 4000000 -n 0` and `./driver_output -b 4000000 -n 4 -l [tas|ttas|mutex|alock|clh]`. The output text files generated are `serial_res.txt, tas_res.txt, tas_fifo.txt, ...`

1.3.1 Mutual Exclusion

The `_res.txt` files store the Counter array from running the program. The Counter array is of size **BIG + 10**, with all entries initialized to 0. In a thread's Critical Section, it increments the Counter and adds one to the slot corresponding to the Counter value in the array. (`lock, incr, record, unlock`) We expect no skips and no duplicates. Therefore, the resultant Counter array should have all BIG entries as 1, and the remaining 10 entries as 0. For example, if BIG is 100, Counter array will be of length 110; we expect the first 100 entries to hold 1, and the last 10 to hold 0. **The reason why we have these 10 extra slots is to check against counting over BIG.**
The script compares the result from each lock with that of SERIAL. This test guarantees the Mutual Exclusion property of the whole program.

1.3.2 FIFO

The `_fifo.txt` files store the local counts of increments each thread achieved when running the program. We expect the FIFO property to hold for two of our fair locks, ALOCK and CLHLOCK. The **Max, Min, Mean, Standard Deviation** of the local counts are printed

for each lock. We expect the Standard Deviation of ALOCK and CLHLOCK to be significantly (this is qualitative only because a quantitative description is difficult) lower than that of the other locks. The statistics differ from run to run, but some representative ones are shown below in **My Own Test**. We observe that the FIFO property hold for ALOCK, CLHLOCK and thus this ensures correctness.

2 Performance Experiment Results

2.0.1 My Own Test

Note: I performed a few trials of the Fairness Test I proposed in the design document and realized that the test is inconclusive. I describe my partial findings as follows. To fulfill this part of the assignment, I instead implemented the TTAS Lock and run both 1. Idle Lock Overhead and 2. Lock Scaling experiments on it.

Originally, I wanted to check whether fairness is related to the number of threads. Specifically, I hypothesize that as the number of threads increases, contention increases and the problem of starvation may become more apparent. However, I did realize as I was writing my design document that fairness is largely dependent on OS scheduling, which is a variable that we cannot control. Even from one trial to another, the variance in number of increases among threads is difficult to predict. Taking the median might not be very representative, and this may render my fairness test not too worthy to conduct. I stated in my design document that I would accept OS scheduling as an explanation if the test turns out inconclusive. From running two trials of BIG=4×10^6, with 4 threads, my result is in the following table. Although this test is inconclusive because of the variability in OS scheduling, we can still draw one useful conclusion from it: **TAS and TTAS are fair locks and may easily result in starvation.** Especially from the bold-face entries, whereas we expect each of the four threads to do 25% of the work, in reality, one thread could achieve $\frac{2840994}{4000000} = 71\%$ or only $\frac{1273}{4000000} = 0.03\%$ of the work. **Mutex is fairer, but still not as fair as the ALock and the CLH Lock, which by design are FIFO locks.**

Note: This is an informal experiment and the data below is collected on a CSIL machine.

		Max	Min	Standard Deviation
Trial 1	TAS	**2840994**	12246	1336098.106
	TTAS	2396307	352158	940018.065
	Mutex	1016344	975702	19113.427
	ALock	1000352	999870	235.200
	CLH	1000102	999948	69.185
Trial 2	TAS	1434878	622706	399820.718
	TTAS	2097538	**1273**	1131209.758
	Mutex	1063239	937516	51388.012
	ALock	1000563	999805	375.441
	CLH	1000330	999863	222.939

Table 1: My Fairness Test

I will discuss my implementation of the TTAS Lock and the tests in the following sections.

2.0.2 Design Choices and Possible Influence on Testing Results

1. Because I want to zoom in the influence of contention only, I start the timer right before spawning the threads. I do not factor the time of initializing the locks into my runtime. **However, in my design document, I assumed that the runtime includes lock initialization. Therefore, all my hypothesis that** ALOCK, CLHLOCK **will have significantly lower throughput than** TAS **because of lock initialization will no longer hold.**

2. I decided to have all my threads stop as soon as the Counter reaches BIG and avoid over-counting at all cost. To achieve this, each of my thread checks for the value of the Counter in its Critical Section; if the Counter has not yet reached BIG, the thread increments the Counter; otherwise, the thread releases the lock and exits. **This adds much overhead to the Parallel version for any of the Locks, resulting in a lower Parallel vs. Serial Throughput Ratio than we'd expect if we allow over-counting. However, I expect the relative trend among Locks to be preserved.**

3. I implemented ALOCK using `PaddedPrimBool_t` provided in `paddedprim.h`. I also replaced the expensive % operation with bitwise operators. This should make ALOCK more efficient.

4. I implemented CLHLOCK by dynamically mallocing nodes as threads enter. I use node-recycling so that there will be no more malloc once there are n nodes. Recycling improves CLH's performance, and we may increase the performance even more by using a pre-allocated pool of nodes since we know n, the number of threads.

Note: I run 1. Idle Lock Overhead and 2. Lock Scaling on both BIG = 10^7 and 10^8. I take the median of five trials to determine each data point.

2.1 Idle Lock Overhead

2.1.1 Original Hypothesis

(Copied from the Design Document) Both ALOCK and CLHLOCK will have a lower throughput than TASLOCK and MUTEX because creating the array or the linked list adds substantial overhead. Because $n = 1$ means contention-free, TASLOCK's only thread will spin on its local copy of the lock `state` variable without ever invalidating its cache; this could possibly lead to an even higher throughput than MUTEX.

2.1.2 Hypothesis for the TTAS Lock

Because $n = 1$ means contention-free, and I do not factor lock initialization time into my runtime, the idle lock overhead only comes from the single thread acquiring and releasing the lock. Therefore, the throughput should be similar for all locks, including TTASLOCK.

2.1.3 Data for BIG = 10^7

For **BIG = 10^7**, I found the serial throughput to be **375080 increments / ms**.

Lock	Throughput Ratio
TAS	0.167120
TTAS	0.155833
Mutex	0.088633
ALock	0.164262
CLH	0.164253

Table 2: Parallel vs. Serial Throughput Ratio where BIG = 10^7

2.1.4 Data for BIG = 10^8

For **BIG = 10^8**, I found the serial throughput to be **377058 increments / ms**.

Lock	Throughput Ratio
TAS	0.155239
TTAS	0.166599
Mutex	0.088282
ALock	0.163699
CLH	0.163701

Table 3: Parallel vs. Serial Throughput Ratio where BIG = 10^8

2.1.5 Observation and Analysis

1. From the Table 2 where BIG = 10^7, there is only small variation in throughput among all the locks except MUTEX. The two most different data points, TAS and TTAS, differ by only 7%. All locks except MUTEX can achieve about 16% of SERIAL's throughput, roughly $375080 \times 16\% \approx 60000$ increments / ms. This means that the idle lock overhead in PARALLEL reduces throughput by 84% compared to SERIAL.

2. We observe that MUTEX has a significantly lower throughput ratio, about 50% lower than any of the other locks. As we don't know the implementation details of MUTEX, we may assume for now that the algorithm it uses makes locking and unlocking expensive, even without contention. As MUTEX is industrial code, it is likely like its deisgn priority is to handle generic situation where contention could be high. In other words, its design might not have optimized for contention-free situations as we tested, but instead focuses on scaling (as we observe in **2. Lock Scaling**.)

3. Table 3 where BIG = 10^8 gives similar result as Table 2. The two most different data points (excluding MUTEX) TAS and TTAS, differ by about 7%.

4. The results make sense, because the relative complexity in a contention-free situation for each of our custom lock algorithm is approximately the same. From what we've learned in class, the major limitation to TAS, TTAS is contention. Without contention, using an idle lock, the amount of overhead (i.e., the cost of a thread reading and writing a bit without frequent cache invalidation) compared to SERIAL is about constant regardless of the lock we use.

2.2 Lock Scaling

2.2.1 Original Hypothesis

(Copied from the Design Document)

1. In short, complex lock algorithms such as ALOCK and CLH scale better than simpler ones like TASLOCK.

2. TASLOCK works well for small n but scales poorly when contention is heavy. As the textbook points out, `getAndSet` lock state can delay all threads because broadcasting the change causes all threads to invalidate their local cache; releasing the lock may be delayed as well. Its curve should start at a high throughput but slopes downward sharply.

3. ALOCK and CLHLOCK scale well with larger n because threads spin on their local variables only. Since the textbook says that CLHLOCK has all the performance advantage ALOCK has, their curves should approximately overlap. Their curves start at a lower throughput than TASLOCK but do not decrease very much as n increases. The generic MUTEX should be similar to ALOCK and CLHLOCK.

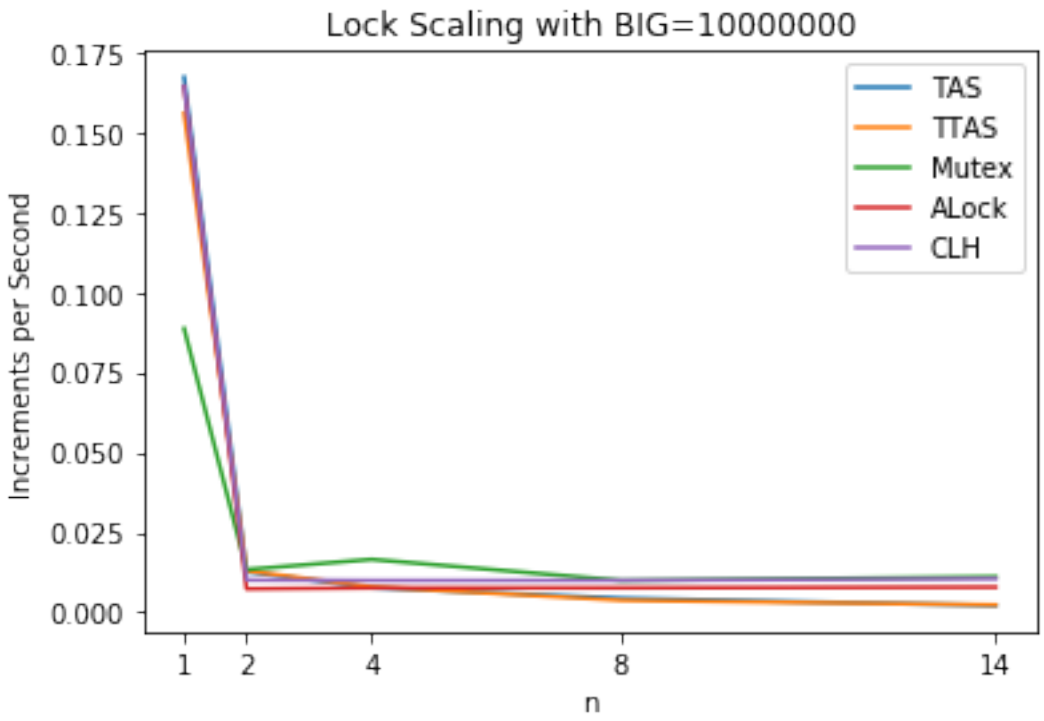
2.2.2 Hypothesis for the TTAS Lock

Unlike TAS which creates busy bus traffic and constantly causes every thread to invalidate its cache, TTAS makes every thread spin on the cached copy and hold onto a valid cache until the lock is released and the state is updated. Therefore, when contention increases, TTAS scales better than TAS. However, in the cases of extremely heavy contention, TTAS's repeated calls to `getAndSet` still create heavy bus traffic. Both ALOCK and CLH should still scale better than TTAS.

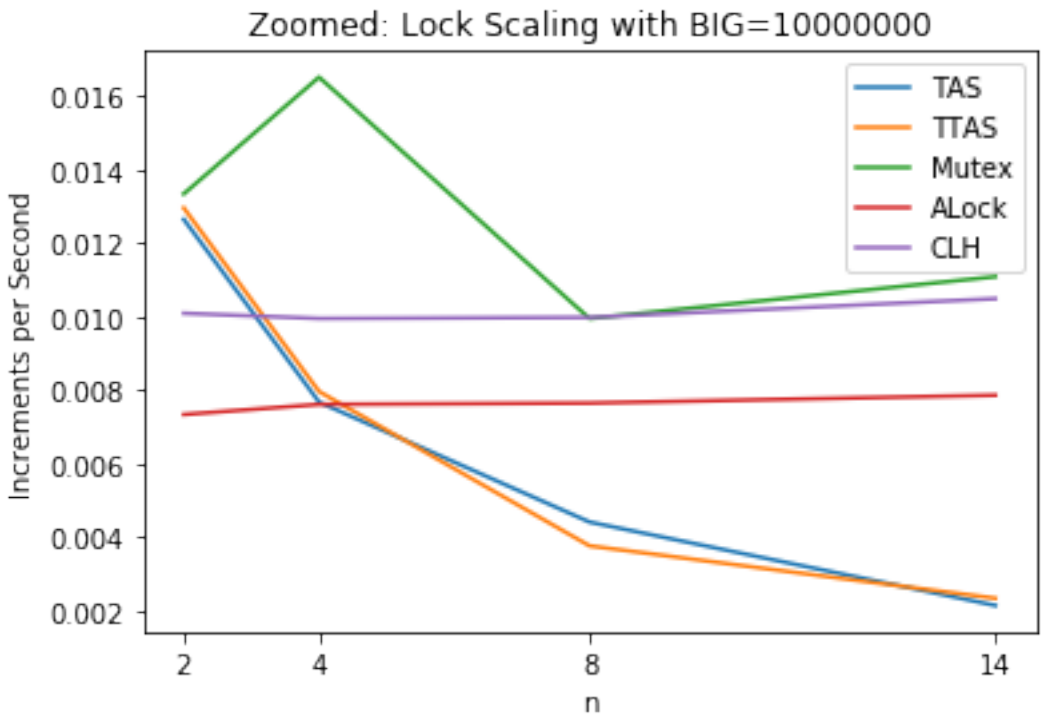
2.2.3 Data and Plot for BIG = 10^7

n	TAS	TTAS	Mutex	ALock	CLH
1	0.167120	0.155833	0.088633	0.164262	0.164253
2	0.012654	0.012965	0.013348	0.007337	0.010095
4	0.007675	0.007964	0.016528	0.007612	0.009953
8	0.004407	0.003757	**0.009951**	0.007648	**0.009983**
14	0.002133	0.002329	0.011092	0.007862	0.010492

Table 4: Parallel vs. Serial Throughput Ratio where BIG = 10^7



As soon as there are more than one thread, the contention significantly decreases the throughput for every lock, including industry-level MUTEX. Therefore, we display a zoomed-in view of $n \in \{2, 4, 8, 14\}$ below.

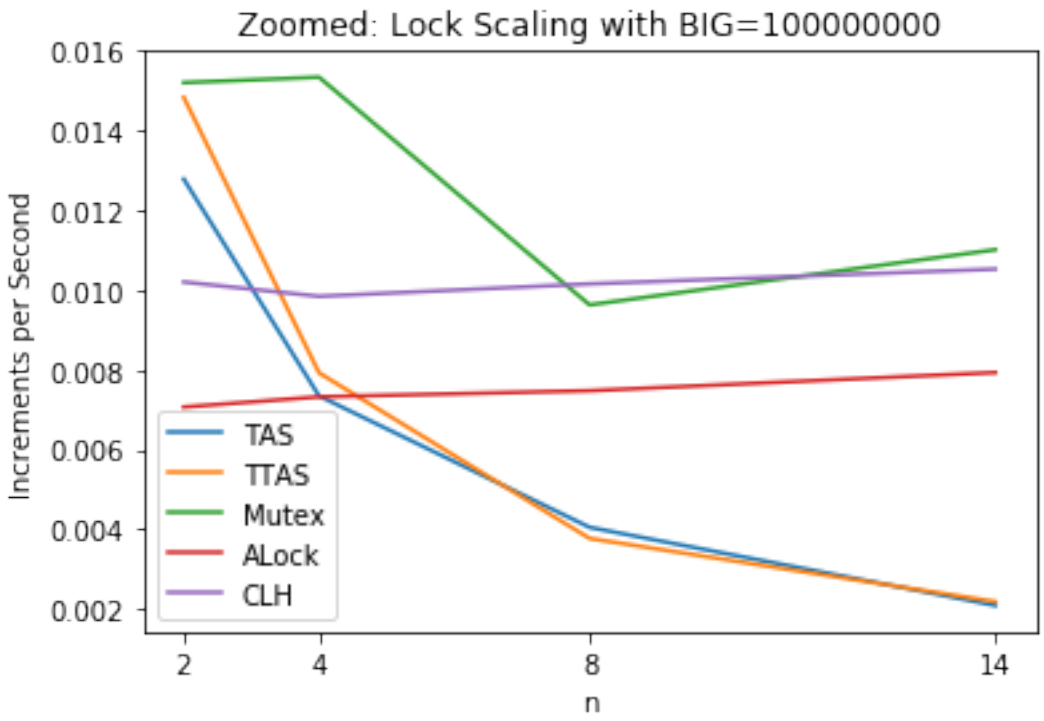


2.2.4 Data and Plot for BIG = 10^8

n	TAS	TTAS	Mutex	ALock	CLH
1	0.155239	0.166599	0.088282	0.163699	0.163701
2	0.012769	0.014813	0.015182	0.007065	0.010194
4	0.007343	0.007920	0.015319	0.007320	0.009835
8	0.004055	0.003778	**0.009621**	0.007473	**0.010146**
14	0.002100	0.002195	0.011003	0.007928	0.010516

Table 5: Parallel vs. Serial Throughput Ratio where BIG = 10^8

Same as the plots above, to observe more distinguishable trends, we display a zoomed-in view of $n \in \{2, 4, 8, 14\}$ below.



2.2.5 Observation and Analysis

1. We find the datapoints for BIG $= 10^7$ and 10^8 to be very similar for the same value of n. This is plausible and expected because unit throughput (# increments/millisecond) should be independent of BIG, as each increments require constant amount of work.

2. For every lock, there is a significant decrease in throughput as soon as there are more than one threads contending for the lock. Quantitatively, in Table 4, from $n = 1$ to $n = 2$, TAS incurs a $\frac{0.1671-0.0127}{0.1671} \approx 92\%$ decrease in throughput; TTAS also incurs a 92% decrease; MUTEX incurs a 85% decrease; ALOCK incurs a 95% decrease; and CLH incurs a 94% decrease. This is plausible as cache invalidation occurs as soon as we introduce contention, and synchronizing via the bus decreases performance.

3. We analyze scaling performance by comparing the **percent decrease** in throughput between $n = 1$ and $n = 14$, as well as between $n = 2$ and $n = 14$ for each lock. We use the data from Table 5. A larger percent decrease means a larger percent increase in overhead due to the contention.

TAS	TTAS	Mutex	ALock	CLH
0.987235	0.985054	0.874855	0.952136	0.936121

TAS	TTAS	Mutex	ALock	CLH
0.831419	0.820363	0.169009	-0.071576	-0.039379

Table 6: Decrease in Throughput, Top: $n = 1$ to 14, Bottom: $n = 2$ to 14

4. As from Table 6, both TAS and TTAS incur over 80% decrease in throughput when scaling from $n = 2$ to 14. On the other hand, MUTEX only incurs a 17% decrease. ALOCK and CLH decrease about 0%, remaining mostly unaffected by scaling.

   The results align with my original hypothesis that ALOCK, CLH, MUTEX **scale better than** TAS, TTAS. As we've learned in class, TAS's overhead increases more than proportionally as contention increases, because of cache coherence; TTAS's overhead increases slightly less than TAS since it decreases the amount of cache invalidation to some extent. ALOCK and CLH's overhead remain almost constant as contention increases, since each thread spins only on a specified, cached location.

5. As in the bold-face entries, CLH performs 5% better than MUTEX for BIG=10^8 with 8 threads. Even when contention is heavy at $n = 14$, CLH and MUTEX differ by less than 5%.

   This result makes sense as each thread only spins on its predecessor's value, and will not interrupt other thread with cache invalidation as it spins. I've also implemented node-recycling and don't need to call malloc again once I have n nodes. We may be able to achieve better performance with CLH if we malloc n nodes in advance and mobilize this static pool.

6. Different from what I'd expect from what we've learned in class, ALOCK is not the best-performing lock among these locks and never once outperforms MUTEX on SLURM. I've done all the optimizations on ALOCK, and found that **on a CSIL machine**, ALOCK sometimes outperforms MUTEX by 5% for BIG = 10^7 and 8 threads. (Median from five trials, ALOCK takes 1240 ms while MUTEX takes 1303 ms.)

   In conclusion, this might be caused by some characteristics of the SLURM machines. For example, they may have cache lines larger than 64 bytes, and our padding won't be large enough to solve cache coherence problems.

7. Different from what I'd expect from what we've learned in class, the performance of TAS and TTAS differ by less than 3% on average. Again, this may be related to the SLURM machines themselves. From five trials **on a CSIL machine**, with BIG =

10^7 and 14 threads, TTAS takes 1832 ms while TAS takes 4014 ms, meaning that TTAS performs 2.2x better than TAS. This informal testing result aligns better with the theory that TTAS handles contention better than TAS.

8. Overall, the results align with my original hypothesis that the complex locking algorithms ALOCK, CLH scale better than TAS and TTAS.